

DESIGN PATTERN DRIVEN DEVELOPMENT
OF MODEL TRANSFORMATIONS

by

HUSEYIN ERGIN

DR. JEFF GRAY, COMMITTEE CHAIR

DR. JEFFREY CARVER

DR. RALF LAEMMEL

DR. RANDY SMITH

DR. EUGENE SYRIANI

DR. SUSAN VRBSKY

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2017

ABSTRACT

Model-Driven Engineering (MDE) is considered a well-established software development approach that uses abstraction to bridge the gap between the problem space and the software implementation. These abstractions are represented by models that make the validation of the real system easier. In MDE, many problems are solved using model transformation, which is a paradigm that manipulates high-level models to translate, evolve, or simulate them. However, the development of a model transformation for a specific problem is still a hard task. The main reason is the lack of a development process where transformations must be designed before implemented. Design patterns provide experiential reuse to software engineers when faced with recurring problems. In the literature, design patterns have been used to generate partially reusable software designs in order to help developers. There are many design patterns focused development methodologies proposed. However, most of them specialize in object-oriented design patterns. Given the various contexts in which design patterns have been applied, model transformations may also benefit from a patterns approach. Although several studies have proposed design patterns for model transformation, there is still no accepted common language to express them or a methodology that places design patterns at the heart of the development of model transformations. Therefore, we created a semi-formal way to describe model transformation design patterns that is independent from a specific model transformation language and described in a practical way that is directly implementable by model engineers. In this dissertation, we present a catalog of 15 model transformation design patterns following a novel uniform template and domain-specific language, DelTa. We elaborate a five-step methodology that guides model engineers in designing solutions to transformation problems by putting the design patterns at the heart of their thought process. We also demonstrate how it is possible to automatically generate excerpts of a model transformation in various languages given a design pattern. We conducted a survey to motivate the need for model transformation design patterns and a user study to validate the usefulness and effectiveness of our methodology to solve problems as model transformations based on design patterns.

DEDICATION

To Tülay, who has always believed in and supported me.

LIST OF ABBREVIATIONS AND SYMBOLS

AD UML Activity Diagrams

AGG Attributed Graph Grammar

ATL Atlas Transformation Language

ATL VM ATL Virtual Machine

AToMPM A Tool for Multi-paradigm Modeling

CD2RD class diagram to relational database diagram

CORBA Common Object Request Broker Architecture

CRUD Create Read Update Delete

DelTa Design Pattern Language for Model Transformations

DSL Domain-specific Language

EMF Eclipse Modeling Framework

ETL Epsilon Transformation Language

FSA Finite State Automate

GoF Gang of Four

GReAT Graph Rewriting and Transformation

GrGen.NET Graph Rewrite Generator

HOT Higher-order Transformation

IDE Integrated Development Environment

IMDB The Internet Movie Database

LCA Lowest Common Ancestor

LHS Left-Hand Side

LMM Language MetaModel

MDE Model-Driven Engineering

MOF Meta-Object Facility

MoTif Modular Timed graph transformation language

MT Model Transformation

MTDP Model Transformation Design Pattern

MTL Model Transformation Language

NAC Negative Application Condition

OMG Object Management Group

PM Pattern Metamodel

PN Petri Nets

PN2SC Petrinets to Statecharts

QVT Query View Transformation

QVT-R Query View Transformation - Relations

RD Rule Diagram

RHS Right-Hand Side

SC Statecharts

TSPEC Transformation Specification

TU Transformation Unit

TUR Transformation Unit Relation

UI User Interface

UML Unified Modeling Language

UML-RSDS UML Reactive System Development Support

VMTS Visual Modeling and Transformation System

XMI XML Metadata Interchange

XSLT Extensible Stylesheet Language Transformations

ACKNOWLEDGMENTS

First of all, I want to thank my wife, Tülay, who has always supported me during my Ph.D. years. Without her, none of these would happen. She always gives me the belief and hope I need in my desperate times.

I would like to thank Dr. Eugene Syriani for his perfect mentorship throughout my Ph.D. He has always pushed me to be a better academician. He is not only a great advisor but also a great teacher, colleague, and friend. This dissertation wouldn't be completed without his precious support and feedback. I will always miss his encouragements.

I would like to thank Dr. Jeff Gray. His ambition for Computer Science has made me revise my vision and dreams about the future. It has been a privilege to see how he has handled a lot of different tasks in his life so smoothly and he is still very successful at the same time. His outreach activities have opened my eyes and let me decide what kind of an academician I want to be.

I would also like to thank my Committee members: Dr. Jeffrey Carver, Dr. Ralf Laemmel, Dr. Randy Smith and Dr. Susan Vrbsky, who have provided feedback in my research and accepted to be on my committee.

I would also like to thank Dr. David Cordes for generously supporting me with research and teaching assistantships during my Ph.D. study.

Finally, I would like to thank all my friends in labs SEC 3419 and 3420. We have always shared ideas and spent time together in order to increase our motivation. They have provided me valuable feedback for many of my talks and presentations.

CONTENTS

ABSTRACT	ii
DEDICATION	iii
LIST OF ABBREVIATIONS AND SYMBOLS	iv
ACKNOWLEDGMENTS	vii
LIST OF TABLES	xiv
LIST OF FIGURES	xvii
1 INTRODUCTION	1
1.1 Problem Statement and Thesis Proposition	3
1.2 Contributions	4
1.3 Outline	5
2 STATE-OF-THE-ART in MODEL TRANSFORMATION DESIGN PATTERNS	6
2.1 Model-Driven Engineering	6
2.2 Model Transformation	8
2.2.1 Model Transformation Approaches	9
2.2.2 Structure of Model Transformation Languages	9
2.2.3 Model Transformation Languages	11
2.3 Design Patterns	16
2.3.1 Object-oriented Design Patterns	16
2.3.2 Design Patterns in Other Programming Paradigms	18
2.3.3 Design Patterns in Computer Science and Engineering	19
2.3.4 Limitations of Design Patterns	19
2.4 Pattern-driven Approaches	20
2.5 Model Transformation Design Patterns	22

2.5.1	Terminology of Reusable Structures	23
2.5.2	Reusable Idioms	23
2.5.3	Design and Refactoring Patterns	25
2.5.4	Classification of Existing Efforts	25
2.6	Language Efforts to Express Model Transformation Design Patterns . . .	28
2.6.1	Rule Diagrams	28
2.6.2	TSPEC	30
2.7	Summary of the Literature Study	30
3	A UNIFIED TEMPLATE FOR MODEL TRANSFORMATION DESIGN PATTERNS	31
3.1	Motivational Survey for a Unified Template	31
3.1.1	Objectives	31
3.1.2	Experimental Setup	32
3.1.3	Participants	32
3.1.4	Results of Transformation Survey	33
3.1.5	Discussion of Transformation Survey Results	34
3.1.6	Threats to Validity	35
3.2	The Unified Template	36
3.3	Design Pattern Language for Model Transformations	39
3.3.1	Abstract Syntax	39
3.3.2	Concrete Syntax	41
3.3.3	Informal Semantics	46
3.3.4	Limitations of DelTa	48
3.4	Comparison of DelTa with Existing Languages to Express Design Patterns	48
3.5	Summary of the Unified Template	49
4	FIXED-POINT ITERATION PATTERN	50
4.1	Running Example	50
4.1.1	Naïve Solution	50

4.1.2	Improved Solution	52
4.2	Similar Problems in Different Domains	54
4.2.1	Equivalent Resistance	54
4.2.2	Dijkstra's Algorithm for Shortest Path	55
4.3	Generalization of the Solution	57
4.4	Promoting the General Solution to a Design Pattern	57
4.5	Summary of Identification of Design Patterns	59
5	MODEL TRANSFORMATION DESIGN PATTERN CATALOG	61
5.1	New Model Transformation Design Patterns	61
5.1.1	Fixed-point Iteration	61
5.1.2	Execution by Translation	61
5.2	Generalized Model Transformation Design Patterns	65
5.2.1	Entities Before Relations	65
5.2.2	Visitor	67
5.2.3	Transitive Closure	69
5.3	Lano et al.'s Model Transformation Design Patterns	72
5.3.1	Object Indexing	72
5.3.2	Top-down Phased Construction	73
5.3.3	Parallel Composition	73
5.3.4	Unique Instantiation	73
5.3.5	Entity Splitting	74
5.3.6	Entity Merging	74
5.3.7	Construction & Cleanup	75
5.3.8	Auxiliary Metamodel	75
5.3.9	Simulating Explicit Rule Scheduling	76
5.3.10	Simulating Universal Quantification	77
5.4	Summary of the Catalog	78

6	DESIGN PATTERN DRIVEN DEVELOPMENT OF MODEL TRANSFORMATIONS	79
6.1	Case Study: Petri Nets to Statecharts	79
6.2	Design Pattern Driven Development Methodology	80
6.2.1	Problem Identification	80
6.2.2	Pattern Selection	81
6.2.3	Adaptation to the Problem	82
6.2.4	Implementation and Refinement	83
6.2.5	Integration	86
6.2.6	Beyond the Process	86
6.3	Automating the Methodology	87
6.3.1	Code Generation with Xpand	87
6.3.2	Higher-order Transformation with ATL	88
6.3.3	Discussion of the Automation Approaches	90
6.4	Tool Support to Guide Model Engineers	92
6.5	Benefits of a Design Pattern-driven Methodology	94
6.6	Summary of the Methodology	94
7	VALIDATION: USER STUDY	95
7.1	Objectives	95
7.2	Experimental Setup	96
7.2.1	GrGen.NET Tutorial	97
7.2.2	The Problem	98
7.2.3	The Methodology	100
7.2.4	Tasks	100
7.2.5	Survey	101
7.3	Data Collection	101
7.4	Participant Selection	102
7.5	Results of the User Study	102
7.6	Threats to Validity	104

7.7	Summary of the Validation	105
8	CONCLUSION AND FUTURE WORK	106
8.1	Summary and Contributions	106
8.2	Future Uses of DelTa	107
8.3	Future Work	108
	REFERENCES	109
	Appendix A MOTIVATIONAL SURVEY QUESTIONS & RESULTS	120
	Appendix B DELTA GRAPHICAL CONCRETE SYNTAX	139
	Appendix C USER STUDY OF THE METHODOLOGY	141

LIST OF TABLES

2.1	Classification of model transformation design patterns. Same patterns with different names are annotated with same letters (e.g., Model visitor and Leaf collector).	26
3.1	The five most used model transformation languages (multiple choice) . .	33
3.2	Design activities performed while planning and solving a model transformation problem (multiple choice)	33
3.3	Comprehension of the design patterns	34
3.4	Comparison of characteristics for design pattern description	38
4.1	Metrics for naïve and improved LCA solutions	53
6.1	Customizing the participants of the design pattern	83
7.1	Effect of the methodology	102
7.2	Task completion ratio	102
7.3	Ratings of the properties	102
A.1	Results of Question A.1.1	120
A.2	Results of Question A.1.2	120
A.3	Results of Question A.1.3	121
A.4	Results of Question A.1.4	121
A.5	Results of Question A.1.5	121
A.6	Results of Question A.1.6	122
A.7	Results of Question A.1.7	122
A.8	Results of Question A.1.8	123
A.9	Results of Question A.1.9	123
A.10	Results of Question A.1.10	124
A.11	Results of Question A.1.11	124
A.12	Results of Question A.1.12	124
A.13	Results of Question A.3.1	125
A.14	Results of Question A.3.5	128

A.15 Results of Question A.3.6	129
A.16 Results of Question A.4.1	129
A.17 Results of Question A.4.4	131
A.18 Results of Question A.4.5	133
A.19 Results of Question A.5.1	133
A.20 Results of Question A.5.2	136
C.1 Results of Question C.1	141
C.2 Results of Question C.2	141
C.3 Results of Question C.3	142
C.4 Results of Question C.4	144
C.5 Results of Question C.5	146

LIST OF FIGURES

2.1	Petri Nets Metamodel and Graphical Concrete Syntax	7
2.2	Meta-layers in MDE	7
2.3	Model Transformation Schema	8
2.4	A Sample Graph-based Model Transformation Rule	10
2.5	MoTif Rules and Scheduling for CD2RD Transformation	12
2.6	GrGen.NET Rules and Scheduling for CD2RD Transformation	14
2.7	Henshin Rules and Scheduling for CD2RD Transformation	15
2.8	ATL Rules for CD2RD Transformation	16
2.9	Rule Diagram Example [45]	29
3.1	DelTa Metamodel	39
3.2	A Sample Pattern in DelTa Graphical Concrete Syntax	42
4.1	Tree instance for LCA problem	51
4.2	Rules for naïve solution	51
4.3	Rules for improved solution	53
4.4	Rules for Equivalent Resistance Problem	55
4.5	Sample input and output electrical circuits model	55
4.6	Rules for Dijkstra’s Algorithm	56
4.7	Sample input and output graph for Dijkstra transformation	56
4.8	Generalization of the solutions with pseducode	57
4.9	Fixed-point Iteration - Structure in DelTa	58
5.1	Execution by Translation - Structure in DelTa	62
5.2	Petri Nets to Statecharts in MoTif	63
5.3	Entities before relations - Structure in DelTa	65
5.4	Rules of Entities before relations pattern in ATL	66
5.5	Visitor - Structure in DelTa	68
5.6	Visitor rules and scheduling in GrGen.NET	69
5.7	Transitive Closure - Structure in DelTa	70

5.8	Transitive Closure rules in AGG	71
5.9	Object Indexing - Structure in DelTa	72
5.10	Top-down Phased Construction - Structure in DelTa	73
5.11	Parallel Composition - Structure in DelTa	74
5.12	Unique Instantiation - Structure in DelTa	74
5.13	Entity Splitting - Structure in DelTa	75
5.14	Entity Merging - Structure in DelTa	75
5.15	Construction & Cleanup - Structure in DelTa	76
5.16	Auxiliary Metamodel - Structure in DelTa	76
5.17	Simulating Explicit Rule Scheduling - Structure in DelTa	77
5.18	Simulating Universal Quantification - Structure in DelTa	78
6.1	Sample Petri Nets model and its Statecharts equivalent	80
6.2	Initialization Rules	84
6.3	AND Reduction Rules	84
6.4	OR Reduction Rules	85
6.5	Finishing Rules	85
6.6	Scheduling of PN2SC Subproblems	85
6.7	Simulation of SC using PN	86
6.8	Code Generation Workflow	88
6.9	Higher-order Transformation ATL Workflow	90
6.10	ATL HOT Snippet for Henshin	91
6.11	Transformation Snippet in Henshin	91
6.12	Activity diagram to generate a model transformation	92
6.13	Design pattern generator tool	93
7.1	Welcome page	96
7.2	IMDB Metamodel and Sample Model Creation Rule	97
7.3	IMDB CreateGroups Rule and The Sample Model Before and After the Transformation	98
7.4	Simplified C and Java Metamodels	98

7.5	The Sample Input and the Output	99
7.6	The Dummy Pattern to Use in Training of the Tool	100
B.1	Model Transformation Design Pattern and Annotation	139
B.2	Transformation Units	139
B.3	Transformation Unit Relations	139
B.4	Pattern Metamodel - Entity	140
B.5	Pattern Metamodel - Relation and Trace	140

CHAPTER 1

INTRODUCTION

Model-Driven Engineering (MDE) is considered a well-established software development approach that uses abstraction to bridge the gap between the problem space and the software implementation [91]. MDE uses models to describe complex systems at multiple levels of abstraction. By using abstraction, there are some benefits [56]: 1) the validation of the correctness of the real system is easier; 2) various implementations for different platforms with respect to the same essential structure and behavior of the system can be produced; and 3) integration and interoperability are handled at a higher perspective rather than platform specific details. In this paradigm, models are first-class elements that represent abstractions of a real system, capturing some of its essential properties. Models are instances of modeling languages that define their abstract syntax (e.g., using a metamodel expressed in a class diagram), concrete syntax (e.g., graphical or textual), and semantics (e.g., operational or denotational by means of a model transformation) [46].

MDE is widely used in industry [11, 39, 51, 80, 92]. The common reasons for MDE adoption are [79]: 1) increasing productivity and shortening development time; 2) improving quality; 3) automation; 4) standardization; and 5) improved communication and information sharing. Baker et al. [11] present 15 years of MDE usage practices at Motorola. They used MDE and Model Transformation (MT) for automatic code generation to transform structured design models into code and automatic test generation to translate scenario-based requirements into conformance test suites. As a result, they realized up to 4X reduction in defects, up to 8X improvement in terms of produced lines of source code and found the defects earlier in the development process. Fleurey et al. [39] report about the huge software migration project accomplished by adopting MDE practices at Sodifrance. They used MDE and MT for the automatic analysis of the existing code base, the reverse engineering of the legacy code to extract high-level models, the transformation

of the generic high-level models to target platform models, and code generation for the target platform. They reported that re-development of the system from scratch would have been more costly than transforming it to a new system using MDE and MT. MDE developers use MTs for various other activities, such as evolving, refactoring, simulating, and manipulating models [75]. These are supported by a plethora of Model Transformation Languages (MTLs) [23], such as Graph Rewrite Generator (GrGen.NET) [42], Henshin [8], and Modular Timed graph transformation language (MoTif) [98], just to name a few.

One of the most important contributions to software design was the Gang of Four (GoF) catalog of object-oriented design patterns [40]. Design patterns help the developers in designing the software before implementation and automatically generating partial code from the design itself. Design patterns are meant to “name, abstract, and identify the key aspects of a common design structure that make it useful for creating a reusable [...] design [40].” This definition has been adapted for graph transformation [2] and, more recently, generalized for model transformation [71]. Design patterns are used in a multitude of software engineering areas, such as in parallel programming [77], finite-state verification [29], but also in other aspects of MDE, namely for Domain-specific Languages (DSLs) [90] and metamodels [21]. A good use of design patterns yields to a better design, however, anti-patterns, which represent bad patterns to apply, also play an important role to prevent common mistakes [17]. Design patterns are also used to communicate about the design, which facilitates design planning, discussion, and documentation [12], given they provide a common vocabulary for design [111].

Design patterns are often used when developing a software project. They are mostly used to manipulate the structural aspects of projects [15, 18, 110]. Design patterns are often high-level solutions to common problems hiding the low-level details. This characteristic makes them ideal to avoid the accidental complexities during development [16]. Researchers have proposed methods and processes that involve automatic code generation from design patterns [15, 85, 88]. To the best of our knowledge, there are no studies that propose a methodology involving model transformation design patterns.

Several design patterns studies have been proposed for model transformation [2, 13, 52, 71]. However, the literature shows no consensus on how to represent these design patterns, especially not in a form independent from existing MTLs, which hampers their reuse and adoption. This also limits the potential to automatically generate concrete model transformation solutions from design patterns, because each design pattern is represented in different languages instead of a unified language. GoF design patterns are described using various Unified Modeling Language (UML) diagrams in order to make the design pattern structure more readable and understandable, which also greatly helps the automatic generation of software [72]. As stated in [95], a design pattern language must be independent from any MTL in which patterns are implemented. Furthermore, a pattern language must be fit to define *patterns* rather than *transformations*. A design pattern language must also be understandable and implementable by a transformation model engineer¹. Additionally, a pattern language must allow to verify if a transformation correctly implements a pattern. Design pattern catalogs evolve over time and new patterns continue to be discovered due to the evolving nature of software and reuse habits of model engineers [9, 26, 41]. Therefore, the design pattern language must not only support the expression of known design patterns, but also be open to define new ones. A first attempt to create a Model Transformation Design Pattern (MTDP) language can be found in [36]. More recently, Lano et al. [71] published a broader study about the topic.

1.1 Problem Statement and Thesis Proposition

Although model transformations are expressed at a level of abstraction closer to the problem domain than code, the development of a model transformation for a specific problem is still a hard, tedious and error-prone task [50]. As witnessed in [45], one reason for these difficulties is the lack of a development process where the transformation must first be designed and then implemented, as practiced in software engineering. The development process should also be supported with the experiences from other model engineers in terms of design patterns. We believe that the design of model transfor-

¹Through the paper, we use “model engineer” in place of “user” or “developer” of a model transformation.

mations can benefit from model transformation design patterns, as it was the case for object-oriented design and in other domains. This will help the model engineer to utilize a plethora of experience and give model engineers the opportunity to generate some portion of the model transformation code automatically, thus eliminating more accidental complexities [16].

The main goal of this dissertation is to improve the quality of model transformations. Our approach makes model engineers explicitly aware of design patterns that can improve their design. In order to facilitate the development, we propose a methodology where model engineers focus initially on using design patterns and the model transformation is partially synthesized from the adaptation of the patterns to the problem at hand. This also helps in reducing efforts in the implementation of model transformation.

1.2 Contributions

The main outcome of this dissertation is a design pattern driven methodology to create model transformations. In order to achieve this outcome, we developed our own model transformation design pattern template to be used as a standardized documentation that is easier to understand by model engineers and has a formal basis for generating model transformation implementations. The contributions of this dissertation are:

1. A survey across the community of model transformation engineers to identify the needs for design patterns and a dedicated unified language to express them.
2. A new unified template to describe model transformation design patterns that consists of characteristics to describe and discuss a pattern along with a domain-specific modeling language, Design Pattern Language for Model Transformations (DelTa), to represent the structure of its solution.
3. The analysis of existing studies in model transformation design patterns that led to identifying 14 unique “real” design patterns in the literature.
4. A process, aided with a tool prototype, to guide model transformation engineers in their design by automatically instantiating patterns in the MTL of their choice,

using template-based code generation and Higher-order Transformation (HOT).

5. A user study to validate the process with real model engineers using the unified template and the tool prototype.
6. Two new design patterns: “Fixed-point Iteration” and “Execution by Translation” as well as the improvement of 13 existing design patterns.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 introduces the necessary background on MDE, MT, and design patterns. We also set the terminology and create a classification for model transformation design patterns. Chapter 3, first, describes the survey that motivated our work on introducing a unified template, and, then, presents the unified template for model transformation design patterns, which includes the definition of DelTa, the design pattern language for model transformations. Chapter 4 illustrates the identification of a new model transformation design pattern: the *Fixed-point Iteration*. In Chapter 5, we outline the catalog of all new, existing, and generalized design patterns presented in the unified template. Chapter 6 presents the design pattern driven development methodology of model transformations on a running example, and the associated tool. In Chapter 7, we report on a user study we conducted to validate the proposed methodology. Finally, Chapter 8 concludes the dissertation and discusses future work.

CHAPTER 2

STATE-OF-THE-ART IN MODEL TRANSFORMATION

DESIGN PATTERNS

This chapter introduces the necessary background on MDE, in particular MT, and design patterns. Then, existing work on design patterns in MT is examined.

2.1 Model-Driven Engineering

MDE [91] is a well-established software development approach that uses *abstraction* to bridge the gap between the problem space and the software implementation. MDE uses models to describe complex systems at multiple levels of abstraction. Models are first-class citizens and represent an abstraction of a real system, capturing some of its essential properties. Models are instances of modeling languages that define their abstract syntax, concrete syntax, and semantics. The *abstract syntax* is the essence of a modeling language, often described by a *metamodel*, which is in the form of entities, relations, and constraints to restrict the modeling language and often expressed by a UML class diagram. The *concrete syntax* defines the graphical or textual notation to represent the elements of the metamodel. Graphical concrete syntax is a mapping of each element in the metamodel to a visual notation. In addition, textual concrete syntax can depend on a grammar (e.g., XText [37]) and set the structure of how the textual model should be represented visually. Figure 2.1 depicts a simplified metamodel for Petri Nets (PNs) on the left, which consists of two elements (Place and Transition) and the relations between them. On the right of the figure, the graphical concrete syntax of each element is presented.

Semantics defines the meaning of the modeling language. The dynamic semantics of a modeling language is often defined by means of a model transformation (either de-

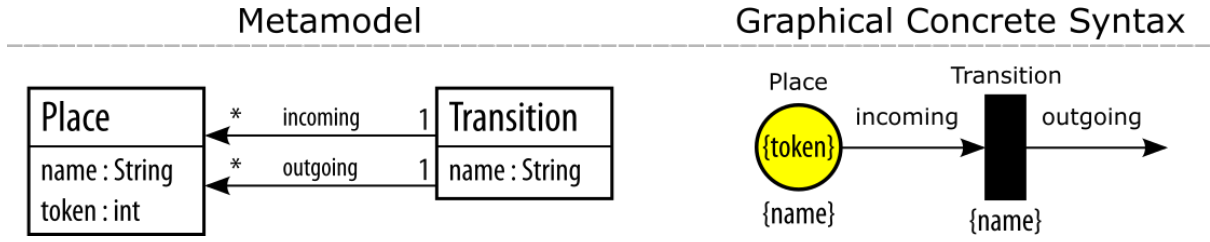


Figure 2.1: Petri Nets Metamodel and Graphical Concrete Syntax

notational or operational). A model expressed in a modeling language *conforms* to its metamodel. Metamodels are also modeled in a modeling language called the “meta-modeling language,” which has a conceptual foundation called a *metametamodel*. In the Object Management Group (OMG) specification, MDE uses the Meta-Object Facility (MOF) as the metametamodel to specify metamodels [44]. The four-layered structure of the meta-hierarchy [61] is depicted in Figure 2.2. M0 level represents the actual system

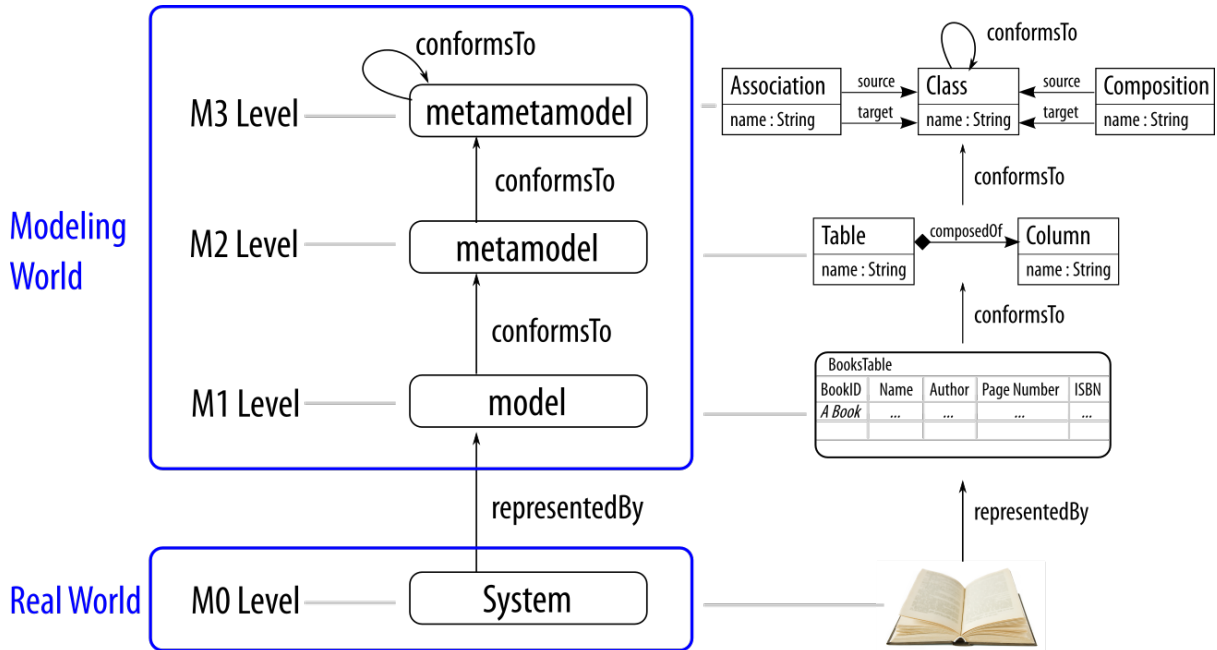


Figure 2.2: Meta-layers in MDE

in the real world with all the data (e.g., a book in real world). The system is represented by a model at the M1 level (e.g., an entry of the book in a relational database), which conforms to a metamodel at the M2 level (e.g., the structure of the relational database). All metamodels also conform to a metametamodel at the M3 level (e.g., the structure how each metamodel should look like), which is often used to define itself.

2.2 Model Transformation

In MDE, the core of the development process consists of a series of transformations over models [89]. Each model transformation conforms to a specific transformation language. Following Jouault *et al.* [54], the model transformation schema in MDE in Figure 2.3 illustrates the key components. **Ma** and **Mb** are models that conform to metamodels

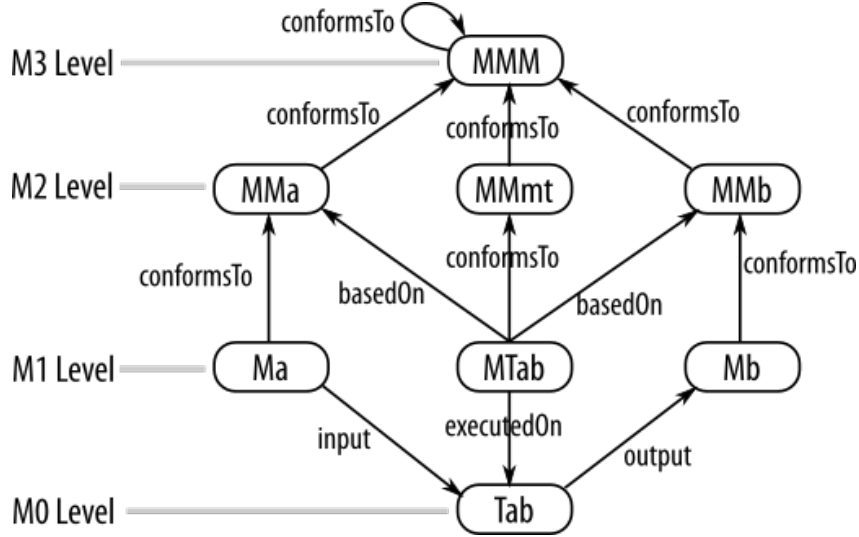


Figure 2.3: Model Transformation Schema

MMa and **MMb**, respectively. **MTab** is a model transformation that is defined for a particular source metamodel **MMa** and target metamodel (**MMb**) and conforms to its own metamodel (**MMmt**). **MTab** is executed as a transformation (**Tab**) at the model level. All three metamodels (**MMa**, **MMb**, **MMmt**) conform to the standard metamodel (**MMM**).

A model transformation is defined as “the automatic manipulation of input models to produce output models, that conforms to a specification and has a specific intent” [75]. These intents play an important role when creating a model transformation. A model transformation intent is a description of the goal behind the model transformation and the reason for using it [75]. For example, for the sake of translational semantics and simulation intents, we have created a transformation from UML Activity Diagrams (AD) to PNs [94].

2.2.1 Model Transformation Approaches

There are two approaches to model transformation [23]: 1) imperative, which is programming the operations of the transformation with explicit instructions on how to transform models; and 2) declarative, which specifies what the transformation should do instead of how by using pre/post-condition rules. The declarative approach consists of: 1) model-to-model languages like Atlas Transformation Language (ATL) [54]; 2) graph-transformation-based languages like MoTif [98]; 3) relational languages like Query View Transformation - Relations (QVT-R) [65]. Model-to-model languages are often out-place, i.e., usually the input model is read-only and the output model is write-only. Graph-transformation-based languages work on graph structures that encode models and allow more flexibility to modify each model element in-place, i.e., the input and output model is the same. Graph-transformation-based approaches can have explicit scheduling structure to let developers define when and how the transformation rules will be applied. Relational approaches usually specify the correspondence between source and target elements, which create the target elements implicitly. In this dissertation, the main focus is declarative approaches that are model-to-model and graph-transformation-based languages.

2.2.2 Structure of Model Transformation Languages

A model transformation mainly consists of source and target languages and their metamodels, transformation rules, and scheduling of the transformation rules.

Transformation Rules are the smallest units of a model transformation. A transformation rule has many different features [23]. The *domain of a rule* defines how a rule can access elements of a model. A rule is a declarative construct that dictates *what* shall be transformed and not *how*. It consists of pre-condition and post-condition patterns. The pre-condition pattern determines the applicability of a rule: it is usually described with a Left-Hand Side (LHS) and optional Negative Application Conditions (NACs). The LHS defines the pattern that must be found in the input model to apply the rule. The NAC defines a pattern that shall not be present, inhibiting the application of the rule. Constraints also can be specified over the attributes of LHS and NAC pattern elements.

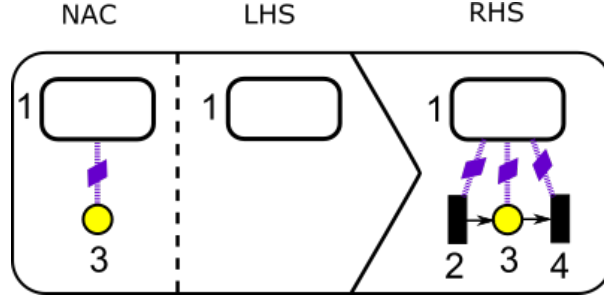


Figure 2.4: A Sample Graph-based Model Transformation Rule

The Right-Hand Side (RHS) describes the post-condition pattern that must be found in the output model after the rule is applied. Imperative actions also can be specified over the attributes of RHS pattern elements. An advantage of using the rule-based transformation paradigm is that it allows to specify the transformation as a set of operational rewriting rules instead of using imperative programming languages. An example of a rule for graph-transformation-based MT with LHS, RHS and NAC parts is depicted in Figure 2.4. This rule is taken from a model transformation that translates an AD model to a behaviorally equivalent PN model [94]. The rule can be read as “if an activity (labeled 1) that is not associated with a place (labeled 3) is found, then create a place and two transitions (labeled 2 and 4), and relate them with temporary trace links (the purple links that have diamonds in the middle).” This rule has a *graphical syntax* using elements from the concrete syntax of the source and target domains (e.g., AD and PN).

Rule Scheduling is an important phase in the development of a model transformation. Scheduling mechanisms determine the order in which individual rules are applied [23]. One can distinguish between implicit and explicit scheduling. When the scheduling of a transformation language is *implicit*, the modeler has no direct control over the order in which the transformation units are applied. On one hand, a transformation language can be *unordered*, i.e., it simply consists of a set of rules. In this case, the order of application of the rules is entirely determined at run-time. It completely depends on the patterns specified in the rules. Applicable rules are selected non-deterministically until none apply anymore. On the other hand, the scheduling of a language can be *explicitly* specified by the modeler. In explicit internal transformation languages, a rule may explicitly

invoke other rules. For example in ATL [54], a matched rule (implicitly scheduled) may invoke a called rule in its imperative part. Finally, in an explicit external transformation language, there is a clear separation between the rules and the scheduling logic. Ordered transformations specify a control mechanism that explicitly orders rule application of a set of rules. The order of the transformation rules can be priority-based, layered/phased, or modeled with an explicit workflow structure. However, most transformation languages are partially ordered. That is, applicable rules are chosen non-deterministically while following the control-flow specification.

2.2.3 Model Transformation Languages

There are many MTLs in the literature. Some examples are: Henshin [8] from Arendt *et al.*, which is a language that operates on models in the Eclipse Modeling Framework (EMF) and has visual syntax, editing functionalities, execution and analysis tools; Graph Rewriting and Transformation (GReAT) [3] from Agrawal *et al.*, which consists of three distinct parts: pattern specification language, graph transformation language and control flow language; Fujaba [60] from Klein *et al.*, which is one of the first tools to do code generation from UML models and UML model generation from code; Viatra2 [106] from Varro and Balogh, which provides a rule and pattern-based language for manipulating graph models by using graph transformation and abstract state machines; and Attributed Graph Grammar (AGG) [100] from Taentzer, predecessor of Henshin [8], which lets graphs be attributed by Java objects and equips graph transformation with computations on these objects.

Each of these languages have a unique set of structure combinations (e.g., different rule and scheduling structure, different directionality). Jouault and Kurtev [55] compared a number of MTLs in terms of transformation scenarios, paradigm, directionality, cardinality, traceability, query language, rule scheduling, rule organization and reflection. Czarnecki and Helsen [23] performed a similar study concentrating on MTs, in general, regardless of the MTLs.

In this dissertation, we heavily use the following four MTLs in the design pattern driven development process we introduce.

MoTif

MoTif [98] is a graph-transformation-based MTL and a short name for “Modular Timed Graph Transformation.” It is integrated into A Tool for Multi-paradigm Modeling (AToMPM) [99] web-based modeling environment. MoTif is an in-place MTL with graphical syntax and has explicit tracing and rule scheduling mechanisms. The scheduling structure is modeled like a UML activity diagram and allows the definition of what happens when a rule is successfully applied or not. MoTif provides flexible elements allowing to model alternative scheduling structures such as loops and conditionals. MoTif also provides various rule application mechanisms such as QueryRule, to check if a rule is applicable or not; ARule, to apply the rule only once for one of the matches of the rule; FRule, to apply the rule for all matches of the rule; SRule, to apply the rule as long as new matches exist.

Figure 2.5 depicts the simple class diagram to relational database diagram (CD2RD) transformation in the MoTif language. The rules’ graphical notation have three compartments: optional NAC, which is represented with the leftmost dashed rectangle; LHS, which is represented with the polygon in the middle; and RHS, which is represented with the rightmost polygon. In the figure, there are three rules: **classMapping**, to map classes to tables; **attributeMapping**, to map attributes of classes to columns of tables; **attrsMapping**, to create the association link between tables and columns based on their class and attribute equivalents. The scheduling structure is depicted in the right part of the figure. The green rectangles are linked to actual rules and represent different application mechanisms

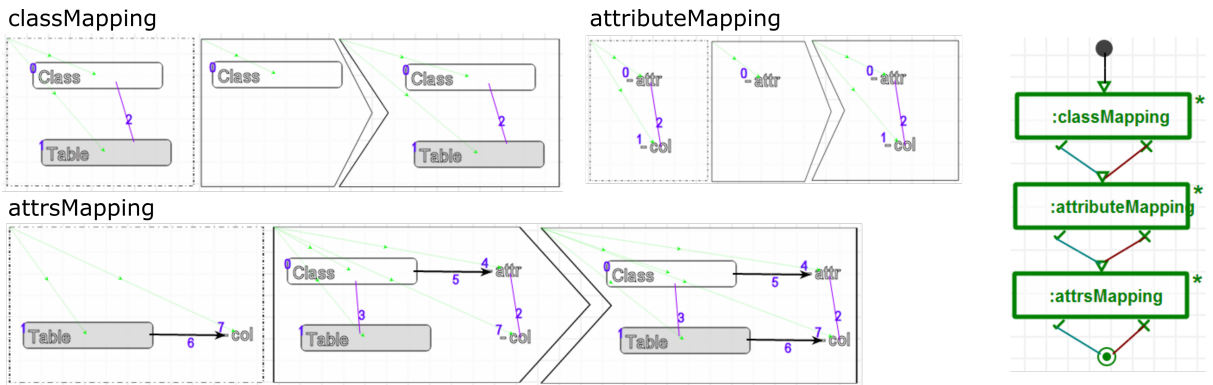


Figure 2.5: MoTif Rules and Scheduling for CD2RD Transformation

mentioned above (i.e., ARule, FRule). The rectangles with an asterisk on the top-right represent an SRule. For this transformation, the rules are scheduled to follow one another, whether they are successfully applied or not, by connecting both output ports of each rule to the input port of the next rule.

GrGen.NET

GrGen.NET [42] is a general-purpose in-place graph rewriting tool. It allows creating model transformations by providing metamodel creation, specifying input and output models, rule designing, and execution of these rules. GrGen.NET provides its own execution environment and processes textual representation of models, metamodels, and transformations. The precondition elements of the transformation rule are directly written inside the rule, basically eliminating a separate part for LHS. The NAC part is defined with a **negative** keyword. Finally, the postcondition is defined with a **modify** keyword allowing the modification of the model as a result of the rule application. In contrast to MoTif, GrGen.NET modifies the elements in the model directly instead of recreating elements via invariants. GrGen.NET provides an explicit scheduling mechanism. This works by referring to the rules by name with some control logic to explicitly denote how they are applied. For example, putting an asterisk after the rule name makes the rule run as long as possible, similar to SRule in MoTif.

Figure 2.6 depicts the CD2RD transformation rules and scheduling of these rules in GrGen.NET syntax. In the figure, there are similar rules for this transformation as described in the previous MoTif language section. The scheduling structure is depicted in the bottom of the figure. Each rule is called with the **exec** keyword, the name of the rule and how to execute the rule (the asterisk).

Henshin

Henshin [8] is a model transformation language that works on attributed graph-transformations. It is an in-place MTL that is an Eclipse plugin and offers a graphical syntax for its rules and scheduling structures. Henshin consumes and produces EMF models. In Hen-

Rules

```
rule classMapping {
  e:Class;
  negative {
    f:Table;
    e -:trace-> f;
  }
  modify {
    f:Table;
    e -:trace-> f;
  }
}

rule attributeMapping {
  e:Attribute;
  negative {
    f:Column;
    e -:trace-> f;
  }
  modify {
    f:Column;
    e -:trace-> f;
  }
}

rule attrsRelationMapping {
  e:Class;
  f:Attribute;
  e -:attrs-> f;
  g:Table;
  h:Column;
  e -:trace-> g;
  f -:trace-> h;
  negative {
    g -:cols-> h;
  }
  modify {
    g -:cols-> h;
  }
}
```

Scheduling

```
exec classMapping*
exec attributeMapping*
exec attrsRelationMapping*
```

Figure 2.6: GrGen.NET Rules and Scheduling for CD2RD Transformation

shin, the rule logic is the same as regular MTL rules. However, in terms of representation, they do not have three compartments as mentioned before (i.e., LHS, RHS, NAC). Instead, Henshin combines these compartments into a single structure and marks elements as to be “created,” “deleted,” and “preserved.” Therefore, this compact notation reduces the number of elements to define in a rule and eliminates the labels that refer to them in the LHS and RHS. Henshin also provides an explicit predefined scheduling structure for common scheduling roles such as sequential rule execution, looping, and conditionals.

Figure 2.7 depicts the CD2RD transformation rules and scheduling in Henshin syntax. The rules have now single compartments. However, each element is marked with stereotypes for distinguishing between LHS, RHS, and NAC. Then, these rules are called sequentially by using a `SequentialUnit`, which basically executes these rules in order with a notation like a UML activity diagram.

ATL

ATL [54] is a model-to-model transformation language and an acronym for Atlas Transformation Language. ATL transformations are outplace, which operate on read-only source models and write-only target models. It provides a textual syntax for transformations and runs on top of Eclipse. ATL rules consist of two clauses: the “from” clause defines


```

rule Class2Table {
  from
    c : Class!Class
  to
    out : Relational!Table (
      name <- c.name,
      -- Columns are generated from Attributes in another rule not
      -- explicitly called here !
      col <- Sequence {key}->
        union(c.attr->select(e | not e.multiValued)),
      key <- Set {key}
    ),
    key : Relational!Column (
      name <- 'objectId',
      type <- thisModule.objectIdType
    )
}

rule ClassAttribute2Column {
  from
    a : Class!Attribute (
      a.type.ocIsKindOf(Class!Class) and not a.multiValued
    )
  to
    out : Relational!Column (
      name <- a.name + 'Id',
      type <- thisModule.objectIdType
    )
}

```

Figure 2.8: ATL Rules for CD2RD Transformation

2.3 Design Patterns

Design patterns “describe a problem which occurs over and over again in our environment, then describe the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [6].” In software engineering, design patterns provide valuable experience in a syntax understandable by software engineers in order to solve various problems faced during development. Good use of design patterns leads to the construction of well-structured, maintainable and reusable software systems [103]. This section explores design patterns in software engineering along with their limitations.

2.3.1 Object-oriented Design Patterns

The most popular study on design patterns was reported by Gamma et al. [40]. In this study, the authors proposed 23 standard object-oriented design patterns. Actually, before

Gamma et al., there were already programming languages that used design patterns without naming them as design patterns explicitly. For example, the model-view-controller structure in Smalltalk-80 is an earlier example of the observer design pattern [63].

For Gamma et al., a design pattern must “name, abstract, and identify the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.” Therefore, the design patterns they created specifically focus on how a previously identified solution can be adopted by a new implementation using an object-oriented programming language. A design pattern is basically described by four main elements: 1) its name, which concisely summarizes the objective of the pattern; 2) the problem it solves, which describes when to apply a pattern; 3) the structure of the general solution, which lists the participants of the pattern: elements, their relationships and roles they play; 4) consequences, which list the advantages, disadvantages, and compromises to be aware of when using the pattern.

An illustration of a design pattern from the book [40] is the “Strategy” design pattern. This design pattern is applied when many related classes differ only according to some specific behaviors. As the solution, the Strategy pattern proposes to delegate the task of the specific behaviors to an interface, encapsulates them in objects, and lets each specific behavior be implemented independently in isolation and in an interchangeable way. The advantages are: 1) the developers’ ending up with a family of related behaviors that are interchangeable; 2) the developers’ eliminating the conditionals by creating a different subclass of the interface for each behavior relying on polymorphism. However, the disadvantages are: 1) the increase in the number of objects created; 2) the notification of the client, who is using the interface, about these different behaviors.

A design pattern should be understandable in order to be applied by any developer to solve the same problem in different contexts. Therefore, Gamma et al. also used the following elements in order to represent a design pattern more precisely: 1) to understand the context better: intent, motivation, and applicability; 2) to make the application easier: structure, participants, and collaborations; 3) to show the pattern in action: implementation, sample code, known uses, and related patterns.

2.3.2 Design Patterns in Other Programming Paradigms

Design patterns are important also in other programming paradigms. There are various studies that have published design patterns for their respective paradigms [7, 10, 66–68, 107]. In most of the studies, the aim for proposing design patterns were: 1) documenting the solution of a common problem in their paradigm; 2) proposing the recipe for solving the common problem; and 3) automating some part of the program development by code generation.

Antoy and Hanus [7] introduced design patterns for functional logic languages. They presented five design patterns in order to solve some general and challenging problems in functional logic languages. They produced a catalog by describing each design pattern with these characteristics: “name,” “intent,” “applicability,” “structure,” “consequences,” “known uses,” and “see also.” In addition, they discussed each design pattern in detail by providing an example implementation in the Curry functional logic language¹.

Laemmel and Visser [68] saw the design patterns as tools for communicating program construction expertise and propose a set of design patterns for functional strategic programming. They produced a catalog of 13 design patterns with these characteristics: “name,” “category,” “also known as,” “intent,” “motivation,” “applicability,” “schema,” “description,” “sample code,” “consequences,” and “related patterns.” They used Haskell code to represent the structure of each design pattern. Strategic programming uses strategies to traverse various terms of the program. In another study by Laemmel et al. [67], they provided a traversal strategy abstraction. Although the core strategies are not specifically mentioned as design patterns, they are primitive elements that solve many problems in strategic programming. They propose four of those primitive elements for a traversal strategy.

Bagge and Laemmel [10] designed a language to represent how to traverse a tree/graph structure. Then, they used this language to express the traversal strategies regardless of the actual implementation language. The language represents a structured pseudocode that can be mapped to an object-oriented language. They described ten traversal strate-

¹<http://www-ps.informatik.uni-kiel.de/currywiki/>

gies that can be used as primitives.

Visser [107] conducted a survey on program transformation rewriting strategies. He first created a taxonomy of the program transformation techniques. Then, he focused on how programs are represented by creating the basic elements to specify program transformations. These basic elements are used as building blocks to represent various program transformation rewriting strategies. Finally, he demonstrated how program transformation rewriting strategies are implemented using these basic elements.

In summary, existing design pattern studies in other programming paradigms focus on: 1) how to partition the target languages into building blocks to create a generic language to represent design patterns; 2) cataloging the design pattern candidates in a systematic format like Gamma et al. [40]; 3) displaying examples and implementations of the design patterns in target languages; and 4) using the building blocks to represent more complex design systems. We adopt a similar approach in this dissertation while creating a catalog of design patterns for model transformations.

2.3.3 Design Patterns in Computer Science and Engineering

Design patterns are very valuable to practitioners to communicate best practices and general solutions to common problems. Therefore, they can be found in various areas of computer science and engineering such as software architecture [20], building Common Object Request Broker Architecture (CORBA) applications [82], real-time systems [28], distributed computing [19], and embedded network systems [109]. In addition, there are design pattern studies for different parts of MDE. In [21], Cho and Gray proposed a list of metamodel design patterns for different problems faced when designing metamodels. There are also design patterns in model transformations [2, 52], which will be investigated in detail throughout this dissertation.

2.3.4 Limitations of Design Patterns

Design patterns are useful in terms of transferring a solution among different implementations by a developer. However, they also have some trade-offs and limitations [1]. The first consequence of using design patterns is trying to apply many of them across the same objects. This hinders the maintenance of the project. It is not clear how many

design patterns are enough in a project. There is also a risk of negative impact on the project by applying too many unnecessary design patterns [22]. However, there are scientific evidences to prove that design patterns help the maintenance of the projects when used appropriately. In a study by Prechelt et al. [84], the authors made two experiments proving that documenting the existence of design patterns in the project significantly reduces the time needed to perform the necessary maintenance to the project.

Design patterns are also applied to eliminate bad design practices [49]. Applying bad design practices (e.g., antipatterns), or applying the good design patterns in a bad way negatively affects the quality metrics of software [17]. A metric may discourage the integration of a design pattern into the project, or vice-versa. For example, the “Bridge” pattern decreases the depth level in a class hierarchy structure [49]. Therefore, its application yields a better design in terms of the depth-of-inheritance quality metric. However, the “Visitor” pattern lets the program traverse the class structure and process each class in an efficient way, while doubling the number of new classes in the system.

Design patterns solve design problems by encapsulating abstractions via objects, focusing on interfaces rather than implementations, enhancing reuse mechanisms, and delegating the tasks to the most appropriate objects [40]. However, these are not valid benefits for all design patterns. According to the study by Khomh and Gueheneuc [57], the “Composite” pattern, which encodes the hierarchy in terms of components and composites, has a positive impact in terms of expandability, reusability, understandability, scalability quality attributes. However, the “Flyweight” pattern, which reduces the memory footprint of large objects, exhibits a negative behavior on most of the quality attributes, except scalability.

2.4 Pattern-driven Approaches

There are many design pattern-driven methodologies in the literature. Focusing on design patterns for development is not a new idea. Because of their high-level representation of a system, patterns are mostly used to generate some part of the code to eliminate low-level implementation details [18]. Existing literature has studied the object-oriented

design patterns and their applications on UML models of real systems.

Boussaidi and Mili [15] propose a three-step approach to incorporate a design pattern into an existing model. In their approach, the steps are: 1) understanding the design problem that the pattern can resolve, 2) recognizing the instance of this problem in the model, and 3) transforming the model to produce the proposed solution. They start with analyzing the patterns and use this approach to evolve a UML class diagram of the problem at hand.

Kim et al. [58] use Query View Transformation (QVT) to manipulate models of the problem based on the design pattern structural (i.e., UML class diagrams) properties and behavioral properties (i.e., UML sequence diagrams). However, they have to create a different set of QVT transformations for each design pattern. The QVT transformation language is used because their approach works on manipulating the EMF Ecore-based models.

Tokuda and Batory [102] create small general-purpose refactoring rules in order to transform existing code. The rules are implemented as functions in C++. Then, design patterns are defined in terms of these rules and applied automatically.

Wang et al. [110] use UML class diagrams extended with stereotypes to apply design patterns on the stereotyped elements. The developer needs to manually identify which elements in the class diagram should be affected by the pattern application by marking them in terms of the design pattern's components. Then, the tool manipulates the stereotyped elements to fit the design pattern description. In order to use this structure, the authors create additional elements in the design patterns so that the components can be easily marked and identified in the actual model.

Schulz et al. [88] propose treating the design patterns as operators rather than building blocks. This idea is similar to "Singleton" design pattern being readily available in modern object-oriented languages. A pattern is considered to be a part of the language. The new operators are applied only on UML class diagram models. Then, Schulz et al. also provide a five-step algorithm for applying a pattern operator into an existing design. The steps are: 1) identification of the problem structure in order to understand the

parts of the model that should be refactored by the design pattern, 2) checking of pre-conditions in order to understand if the interfaces are suitable for a refactoring operation, 3) parametrization of the design pattern according to the problem structure, because design patterns hold generic aspects and the problem has the application-specific details, 4) reorganization of the context to update the affected interfaces to provide same functionality, and 5) checking of post-conditions that usually consist of informal descriptions of the system such as “some specific implementation should be interchangeable.”

Radeke et al. [85] propose a tool for User Interface (UI) development so that the developers can focus on high-level design instead of dealing with low-level implementation details. The developers can choose from a pool of UI patterns and apply them to the actual system. The authors also propose a four-step process for pattern application: 1) identification of where to apply a pattern, 2) selection of the appropriate pattern for the places identified in the previous step, 3) instantiation of the pattern as a concrete structure so that it is customized and adapted to the problem at hand, and 4) integration of the pattern instance back to the original model so that it is connected and collaborating well with other components.

In summary, most of the existing design pattern-driven approaches try to implement an automatic generation feature from the design patterns and integrate them back to the original model. The authors in these studies also tend to guide developers on how to accomplish this task using a step-by-step process. In general, a pattern-driven approach should at least consist of: 1) manual intervention from the developer in order to understand the places to apply a pattern, 2) customization of the general-focused, high-level pattern with respect to the problem at hand, 3) generation of either a partial or a full model based on the design pattern, and 4) integration of the smaller generated pieces back to the original model.

2.5 Model Transformation Design Patterns

In the following, after we define the terminology, we discuss initial works on design patterns for model transformation. We also classify the existing design patterns according

to our terminology.

2.5.1 Terminology of Reusable Structures

We note that not all model transformation design patterns proposed in the literature should be considered as a design pattern; some are reusable idioms, even refactoring patterns, and others are specific to a particular MTL, though some can be generalized as a design pattern. A *design pattern* should be language-independent and applicable in various MTLs, whereas a *reusable idiom* is only applicable within the context of a specific MTL. Additionally, a design pattern is a reusable solution that should be applicable whether the transformation on which it will be applied exists (e.g., to optimize the transformation) or not (e.g., to design the transformation from the beginning). This leads to the difference between a design pattern and a *refactoring pattern*, where the latter is only applicable when there is an existing solution. We distinguish between the following concepts:

- **Reusable Idioms** are language-specific structures that are reusable within a single MTL [2]. They are often presented as a feature built into the MTL, e.g., *multiple matching pattern* in ATL [13].
- **Design patterns** are language-independent solutions to a class of problems and are applicable to any MTL [40], e.g., *mapping pattern* [52].
- **Refactoring patterns** are language-independent structures that improve an existing transformation according to some quality criteria [78], e.g., *introduce rule inheritance* [71].

2.5.2 Reusable Idioms

Initial studies on reusable structures proposed useful idioms that are specific to model transformation languages: GReAT [2], QVT-R [52], ATL [13], and Visual Modeling and Transformation System (VMTS) [74]. Therefore, they should not be considered as design patterns for model transformation, but reusable idioms in a specific MTL. Additionally, they are all defined as model transformations, rather than patterns, and use specific input

and output metamodels. Therefore, it is not clear how to reuse these patterns for different application domains or in other languages.

The first work that proposed reusable idioms for model transformation was by Agrawal et al. [2]. They defined the *transitive closure* pattern to calculate transitive closure of the links in a graph structure. The *leaf collector* pattern traverses a hierarchical tree to find and process all leaves. The *proxy generator* idiom is not a general design pattern, since it is specific to languages modeling distributed systems where remote interactions to the system need to be abstracted and optimized. These idioms are defined in the GReAT language.

Iacob et al. [52] defined five other reusable idioms for outplace transformations, where the input model is not modified during the transformation and the output is a separate model. The *mapping* idiom first maps entities and then relations. Because it is described using QVT-R, we consider it as an implementation of the entity-relation mapping pattern described in [36]. The *refinement* idiom proposes to transform an edge into a node with two edges in the context of a refinement transformation, so that the target model contains more detail. The *node abstraction* idiom abstracts a specific type of node from the target model while preserving its original relations. The *flattening* idiom removes the composition hierarchy of a model by replacing the containment relations. The *duality* idiom is not a general design pattern, because it is specific to data flow modeling languages that convert edges to nodes in the flow.

Bézivin et al. [13] mined ATL transformations and discovered two reusable idioms. The *transformation parameters* idiom suggests to model explicitly auxiliary variables needed by the transformation in an additional input metamodel, instead of hard-coding them in ATL helpers. The *multiple matching* idiom shows how to match multiple elements in the *from* part of an ATL rule. Newer versions of ATL already support this feature, which makes this idiom obsolete.

Levendovszky et al. [74] proposed domain-specific reusable idioms for model transformation as well as different DSLs. In their approach, they defined reusable idioms using a specific MTL, VMTS, where rules support metamodel-based pattern matching. They

proposed two reusable idioms: the *helper constructs in rewriting rules* idiom explicitly produces traceability links, and the *optimized transitive closure* idiom which is similar to the one from Agrawal et al. [2].

2.5.3 Design and Refactoring Patterns

Lano et al. [71] provided the most comprehensive model transformation design pattern study. This study proposes a total of 29 patterns classified in five categories.

Rule modularization patterns are meant to “improve the structural quality, flexibility, and maintainability of model transformations” [71]. These include the *phased construction* pattern to decompose a transformation into phases. Optimization patterns are “concerned with improving the efficiency of a transformation” [71]. These include the *decomposing complex navigations* pattern to simplify expressions. Model-to-text patterns deal with code or text generation from models. These include the *model visitor* pattern to generate text in a systematic way. Expressiveness patterns aim to overcome MTL restrictions by providing alternative solutions. These include the *simulating universal quantification* pattern to replace for-all conditions with a double negation. Architectural patterns are “concerned with organizing systems of transformations in order to enhance the modularity, verifiability, and efficiency of these systems” [71]. These include the *phased model construction* pattern to construct the target model using separate input models.

The authors also explained relations between and combinations of these patterns, and how to select patterns according to transformation intents. Lano et al. used a subset of transformation intents such as refinement, abstraction, and migration. However, a complete list of model transformation intents is also available [75].

They described each pattern with the following fields: summary, application conditions, solution, benefits, disadvantages, applications and examples, and related patterns. Each field is used to explain the pattern precisely.

2.5.4 Classification of Existing Efforts

In Table 2.1, we classify existing pattern structures in model transformations according to the terminology we have provided in Section 2.5.1. Design patterns marked with

‘*’ are classified as reusable idioms, but they can be promoted to design patterns if more MTLs support the features they depend on.

Most patterns from the *rule modularization* category are considered design patterns according to the above definitions. *Phased construction* (see Section 5.3.2) is a general pattern that encompasses *structure preservation*, *entity splitting* (see Section 5.3.5), *entity merging* (see Section 5.3.6) and *map objects before links* (see Section 5.2.1). These patterns require the transformation to be applied in phases. For example, in *Map objects before links*, objects need to be mapped in the first phase and links in the second phase. *Auxiliary metamodel* (see Section 5.3.8) is a pattern to support temporary elements that do not belong to source or target metamodels. *Construction and cleanup* (see Section 5.3.7) again show some similarity to *Phased construction*, since the construction phase is clearly separated from the cleanup phase. *Parallel/serial composition* (see Section 5.3.3) requires rules be independent from each other in terms of reading and writing attributes. Rule inheritance is another feature that may not be supported by many MTLs, but a useful structure when it comes to eliminating redundancy in the rules.

Expressiveness patterns are useful when a language lacks support for a specific feature. *Simulating universal quantification* (see Section 5.3.10) provides a for-all support by using

Study	Design Pattern	Refactoring Pattern	Reusable Idiom
Lano [71]	Phased construction, Structure preservation, Entity splitting, Entity merging, Map objects before links ^a , Parallel composition, Auxiliary metamodel ^b , Construction and cleanup, Recursive descent, Introduce rule inheritance, Unique instantiation, Object indexing, Model visitor ^c , Simulating universal quantification, Simulating explicit rule scheduling, Replace fixed point by bounded iteration*, Implicit copy*, Replace abstract syntax by concrete syntax*	Replace explicit calls by implicit calls, Omit negative application conditions, Decompose complex navigations, Restrict input ranges, Remove duplicated expression evaluations	
Bezivin [13]	Transformation parameters ^b		Multiple matching
Levendovsky [74]	Optimized transitive closure ^d , Helper constructs in rewriting rules*		
Agrawal [2]	Transitive closure ^d , Leaf collector ^c		Proxy generator
Iacob [52]	Mapping ^a , Node abstraction*	Refinement, Flattening	Duality
Ergin [36]	Entity-relation Mapping ^a , Transitive Closure ^d , Visitor ^c , Fixed-point iteration		

Table 2.1: Classification of model transformation design patterns. Same patterns with different names are annotated with same letters (e.g., Model visitor and Leaf collector).

double-negations and simulating explicit rule scheduling for languages that have implicit scheduling.

Among optimization patterns, *unique instantiation* (see Section 5.3.4) checks for an existing element with the same properties before creation, and *Object indexing* (see Section 5.3.1) helps to refer to elements in different rules by using a key.

The patterns *recursive descent* and *model visitor* (see Section 5.2.2) work on hierarchical structures and require processing of nodes in these structures while traversing them.

Architectural patterns are generalizations of rule-level patterns to organize transformations. In Lano et al., the structure is provided using activity diagrams. Although architectural patterns are useful, they do not serve the purpose of representing design patterns to detect and instantiate them because they are too general, which adds much complexity for achieving this task. Therefore, we have excluded architectural patterns from design patterns we considered.

Replace fixed point by bounded iteration is a language-specific feature, for example using FRules instead of SRules in MoTif [98]. FRule matches the rule’s pre-condition at the beginning, therefore executing the rule for a fixed number of times, whereas SRule matches the pre-condition in each iteration and works as long as the rule is applicable on the model. *Implicit copy* and *replacing abstract by concrete syntax* are language-specific patterns. However, they provide useful features to have in an MTL, thus can be generalized and promoted to design patterns.

We identified five patterns from Lano et al. [71] that are in fact refactoring patterns: *Replace explicit calls by implicit calls*, *Omit negative application conditions*, *Decompose complex navigations*, *Restrict input ranges*, and *Remove duplicated expression evaluations*. All of these patterns require a transformation to exist (as stated in their application condition) in order to optimize specific features of the transformation. In addition, some of the patterns proposed by Lano et al. are anti-patterns (i.e., how not to do things), such as entity merging and entity splitting.

Bezivin’s [13] *transformation parameters* is a design pattern which we generalize to

“auxiliary metamodel.” *Multiple matching* is a feature of the ATL transformation language, therefore it is a reusable idiom.

Optimized transitive closure by Levendovszky [74] is a design pattern that can be identified in some other studies [2, 36] and also in this dissertation. *Helper constructs in rewriting rules* is considered a design pattern, because creating traceability links can be reused in various MTLs.

Agrawal et al.’s [2] *leaf collector* is a visitor design pattern and proxy generator idiom is considered a reusable idiom because it is specific to distributed systems modeling languages.

Iacob et al.’s [52] *mapping* idiom is identified as a design pattern in other studies as well [36, 71]. *Node abstraction* can be carried out to be a design pattern because it proposes a generic solution to identify some specific nodes. *Refinement* and *flattening* requires some input transformation and optimizes the structure of the rules, therefore they are refactoring patterns. *Duality* is a reusable idiom to convert edges to nodes in a data flow.

Finally, all patterns in Ergin et al. [36] are design patterns specifically crafted from existing studies for this purpose.

2.6 Language Efforts to Express Model Transformation Design Patterns

There are not many studies that focus on expressing model transformation design patterns, which is also another problem in the field. All existing model transformation design pattern studies reviewed in Section 2.5 use a specific MTL to represent their design patterns. Nevertheless, we have found two studies that propose a dedicated language for representing model transformation design patterns.

2.6.1 Rule Diagrams

Guerra et al. [45] proposed a collection of languages to engineer model transformations and, in particular, for the design phase. They propose a formal workflow that keeps traces between the different phases in the collection. Each phase involves the production of necessary models conforming to the respective language. Rule Diagrams (RDs), which

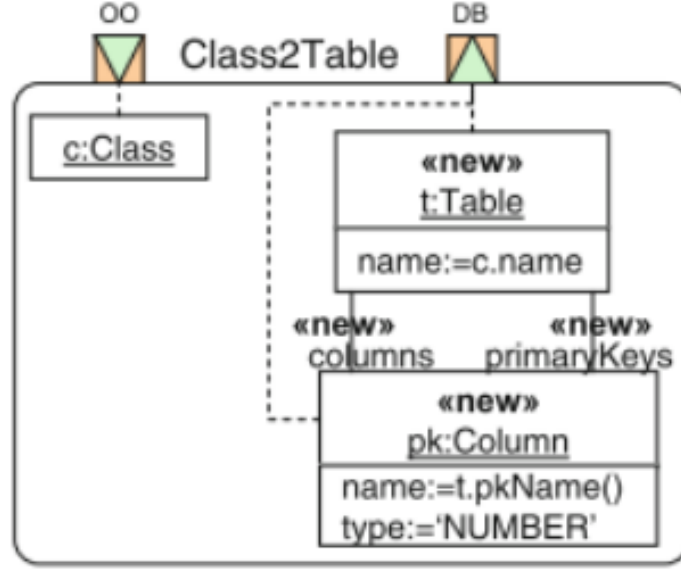


Figure 2.9: Rule Diagram Example [45]

represent the language that automatically produces the implementation of the transformation, are used to describe the structures of the rules and their tasks in the low-level implementation phase. RDs are defined at a level of abstraction that is independent from existing MTLs. Rules focus on mappings rather than constraints and actions (i.e., LHS and RHS). The metamodel of a RD strictly specifies that the transformations are based on mapping models received from the mapping phase of the collection. Therefore, there needs to be at least two metamodels involved in the transformation to map with each other. They specify designs for both unidirectional and bidirectional rules.

The scheduling of rules allows for sequencing and branching in alternative paths based on a constraint. The execution flow of RD supports sequencing rules, branching in alternative paths based on a constraint which is similar to decisions in UML activity diagrams, or non-deterministically choosing to apply one rule. They also allow rules to explicitly invoke the application of other rules. RD is inspired from QVT-R [65] and Epsilon Transformation Language (ETL) [62] and is therefore more easily implemented in these languages. Figure 2.9 depicts an example of a rule diagram that maps a class to a table in a class diagram to relational database diagram transformation. In this rule, the newly created elements are marked with “new,” such as the table and its corresponding primary key, the column.

2.6.2 TSPEC

Lano et al. [71] proposed Transformation Specification (TSPEC) as the language to describe the structure of design patterns. The purpose of TSPEC is to formalize whole transformations. The authors apply formal analysis techniques by using the TSPEC representation of the transformation. TSPEC provides precise definition of the transformations in order to analyze the cost of their understandability and ease of implementation. However, the authors use the UML Reactive System Development Support (UML-RSDS) [69] language to implement their specifications.

TSPEC uses mappings with constraints to represent rules in a transformation, by incorporating another separate metamodel, named the Language MetaModel (LMM), to represent the languages on which the transformations operate upon. Listing 2.1 depicts an example TSPEC transformation specification of a similar transformation shown in Figure 2.9. It transforms class diagram entities to tables and attributes of classes to columns of tables.

Listing 2.1: TSPEC Example [71]

```
for each c : Entity
  create t : Table satisfying t.name = c.name and t.schema  $\simeq$  c.package
for each a : Attribute
  create cl : Column satisfying cl.name = a.name and cl.table  $\simeq$  a.owner
```

2.7 Summary of the Literature Study

In this chapter, we introduced the terminology to distinguish between a reusable idiom, a design pattern and a refactoring pattern by analyzing and classifying the existing patterns in the literature. This chapter points out that there is no consensus in the literature how to name or represent the model transformation design patterns. The authors usually choose to use the model transformation language they are using in practice for this task. There is therefore a need for a unified structure to represent the model transformation design patterns in an MTL agnostic way. However, this should be supported with a user study for further insights about the topic from the community, which we conduct in the next chapter.

CHAPTER 3

A UNIFIED TEMPLATE FOR MODEL TRANSFORMATION DESIGN PATTERNS

In this chapter, we present a unified template to represent model transformation design patterns. We first investigate the need for a unified template with a user survey. One of the most important characteristics to describe a design pattern is the structure of the solution it proposes. Therefore, the remainder of the chapter focuses on DelTa, a language dedicated for this purpose.

3.1 Motivational Survey for a Unified Template

In this section, we describe the survey that was conducted to motivate the need for a language to express model transformation design patterns. We believe this is needed because Chapter 2 shows no consensus on how to represent model transformation design patterns.

3.1.1 Objectives

In this study, we were specifically interested in answering the following questions.

RQ1 “Is there a need for a common language to describe model transformation design patterns?” Chapter 2 clearly shows that the authors who propose reusable idioms or design patterns tend to use an existing MTL for representation. We need to identify if a new language may be adopted by the community.

RQ2 “Is DelTa an appropriate candidate to describe model transformation design patterns?” We have created an initial prototype¹ of a language by analyz-

¹Although, we have used an initial prototype for this survey that is published in [36], the final version of DelTa is presented in later chapters of this dissertation.

ing and synthesizing existing MTLs in order to understand if our language would be a fit for future uses.

RQ3 “How can a model transformation design pattern improve the implementation of model transformations?” Given the limitations and benefits of design patterns outlined in Chapter 2, we wanted to understand how model engineers use design patterns when implementing model transformations.

3.1.2 Experimental Setup

We prepared an online survey² with a total of 22 questions. We also supported some of the questions with a feedback textbox requesting the reason behind the answer. The survey was closed to selected participants only. We used the Qualtrics³ software to collect and analyze the results. The full survey is in Appendix A.

There was no time limit to complete the survey and participants had access to any resource they needed. The survey consisted of four blocks of questions or explanations. The first block had 10 questions and focused on background information about the participants, such as familiarity with design patterns, software design, and model transformation. The second block had no questions, but introduced a preliminary version of DelTa [36] and its purpose, along with referring the participant to another document that shows DelTa’s concrete syntax in detail. In the third block, we tested the ability of participants to understand and interpret two design patterns in which the structure is represented in DelTa: a simple one (entities before relations) and a more complex one (fixed point iteration) from [36]. Here, the level of complexity is relative to the number of constructs used in the design pattern. We asked four questions for each of the design patterns. The final block has four questions and finalizes the information regarding the three research questions.

3.1.3 Participants

We selected participants from attendees at conferences and workshops where model transformation is a main topic of interest (e.g., International Conference on Model Trans-

²<http://tinyurl.com/DelTaSurvey2015>

³www.qualtrics.com/

formation, Transformation Tool Contest). Additionally, the participants must have developed at least one model transformation in the past. A total of 23 participants completed the survey. According to the background question results, 95% of the participants developed software for academic purposes and an average of 27% of their development time focused on the design phase. 44% of the participants used hand sketches for designing and 26% used a UML tool for software development. All participants were familiar with object-oriented design patterns. On average, 39% of their development time included model transformations.

3.1.4 Results of Transformation Survey

Language	Used by
ATL [54]	36%
ETL [62]	23%
Henshin [8]	9%
MoTif [98]	9%
QVT-OM [65]	9%

Table 3.1: The five most used model transformation languages (multiple choice)

Table 3.1 shows the most popular model transformation languages used by the participants. They could choose multiple selections from 11 languages we proposed and another field where they could enter the name of a model transformation language not listed. Table 3.2 lists the design activities performed by the participants while planning and solving a model transformation problem. Activities included a range of options from hand sketches to the tool’s built-in support for design. Hand sketches (64%) are still the most used method when planning and solving a model transformation. However, some languages (e.g., MoTif, GrGen.NET) have dedicated Integrated Development Environ-

Model Transformation Design Activity	Performed by
Hand sketching	64%
Directly implement without designing first	18%
Think of solution in mind	14%
Use image editing tools	14%
Tool used has support for design	9%

Table 3.2: Design activities performed while planning and solving a model transformation problem (multiple choice)

Comprehension	Design Pattern 1	Design Pattern 2
Understand the design pattern	91%	86%
Can see how to implement it	68%	68%

Table 3.3: Comprehension of the design patterns

ments (IDEs) that assist the model engineers design the model transformations. Table 3.3 depicts the participants’ understanding of design patterns and how to implement them in their language. Finally, 82% of the participants agreed that it is appropriate to design the solution using a specific notation first, before implementing the transformation, 68% agree that it is useful to have a language dedicated to designing model transformations, analogously to UML for object-oriented programs. The complete results of this survey are in Appendix A.

3.1.5 Discussion of Transformation Survey Results

RQ1: 64% of the model engineers resort to hand sketches when planning the solution to a problem that will use a model transformation (Table 3.2). The main reason reported is due to the lack of tools to design model transformations. A large majority (68%) agree that a language for this purpose, such as DelTa, is needed.

RQ2: Although 12 out of 22 participants stated that DelTa is an appropriate design pattern language (7 participants were neutral about DelTa), they have almost unanimously understood both patterns well. Furthermore, 59% stated that patterns described in DelTa are easily implementable in their favorite model transformation language. We also directly asked about the understandability and implementability of the design patterns with a 5-point scale, from strongly agree to strongly disagree. The results are in Table 3.3. It is important to note that the survey used an earlier version of DelTa presented [35], but in graphical concrete syntax. Following the comments gathered from this survey, we incorporated several of the useful improvements suggested by the participants (e.g., removal of transformation block, converting *random* to *choice*). These changes we incorporated from the participants led to the version of DelTa presented in this dissertation. Three participants stated they did not think DelTa is an appropriate language. Two of them were suspicious about the benefits of introducing a new language, given the

already many existing model transformation languages. However, DelTa is not a model transformation language, but a language for describing design patterns that abstracts concepts present in existing model transformation languages. The other participant was worried that DelTa may not express complex transformations. However, DelTa does not aim at defining complete transformations, but at restraining how a transformation should be implemented.

RQ3: Besides regular improvements of the transformation code (such as readability, understandability, optimization), a model transformation design pattern helps the model engineers to change their current behavior. There is still a large majority of model engineers doing hand sketches to design a model transformation before implementation (64%). The model engineers tend to use a tool if it exists. Also, they think DelTa is an appropriate language to express model transformation design patterns. Therefore, a tool with a semi-automatic generation from DelTa design patterns to model transformation solutions in a model transformation language should be helpful in the implementation process. In addition, model engineers think it may help to document the knowledge in the domain and understand the complexity of the transformation before implementation.

3.1.6 Threats to Validity

There are various threats to the validity of this survey. Threats to internal validity include the need to understand DelTa before answering the survey questions about design patterns. Although DelTa's aim is to simplify and increase the understandability of the design pattern structure, model engineers are suggested to read the paper in which DelTa is introduced [36] and a reference guide to understand graphical syntax of DelTa as depicted in later chapters of this dissertation. We have tried to eliminate this threat by making the introduction as clear as possible in the latter document.

Threats to external validity include the experience level of the model engineers. All our model engineers are from an academic background, which removes the effect of the study in an industry setting. One other threat is the number of participants and how far we can generalize the results.

3.2 The Unified Template

According to the feedback gathered in the survey in the previous section, although DelTa is a good candidate to describe a design pattern, it is not sufficient alone. A more complete description similar to GoF [40] design patterns was suggested. As shown in Chapter 2, there is no agreement on how to represent model transformation design patterns. Various studies have used different characteristics to represent a design pattern (e.g., applicability, benefits, and structure). Table 3.4 depicts the correspondences between existing proposals for model transformation design pattern templates. In addition, there is no common language that provides the structure of a model transformation design pattern, analogous to how UML is used in representing the structures of object-oriented design patterns. Therefore, we propose to unify the existing design pattern representation templates and improve them with the appropriate language (i.e., DelTa) to define the structure of each design pattern. The middle columns in Table 3.4 show which characteristics are used in different studies to represent design patterns, along with their equivalents with the template used in GoF in the last column. After analyzing all different notations and templates used in existing approaches, we propose to merge the respective characteristics as a unified template shown in the first column. They are mostly influenced by Lano et al. [71] since it was the most complete and thorough template in the literature. In the unified template, a design pattern consists of the following characteristics:

- **Summary:** a short description of the design pattern that usually gives the outline of the other characteristics in a few sentences.
- **Application Conditions:** pre-conditions on the context of pattern use. The conditions can be either pre-conditions on the metamodel or constraints over the transformation.
- **Solution:** generic solution to the problem the design pattern addresses. The structure of the solution is expressed in DelTa.
- **Benefits:** advantages of applying the design pattern. The benefits can either

be measurements with respect to some quality criteria or improvements on some features of the transformation.

- **Disadvantages:** pitfalls of applying the design patterns. The disadvantages can be measurements with respect to some criteria.
- **Examples:** concrete application of the design pattern in a real context. The example is implemented in a specific model transformation language.
- **Implementation:** discussion providing guidelines and hints on how to implement the design pattern in various transformation languages.
- **Related patterns:** correlation of the pattern with other patterns. This relation may be specialization, generalization, sequence, grouping, alternatives, or others.
- **Variations:** different versions of the pattern. This can either be with small tweaks or other alternative representations of the pattern.

Unified Template	Bezivin [13]	Levendovsky [74]	Agrawal [2]	Iacob [52]	Lano [71]	Ergin [33]	GoF ing	Meaning
Summary	Motivation	Motivation	Motivation	Goal Motivation	Summary	Motivation	Intent Motivation	
Application Condition		Applicability	Applicability	Applicability	Application Conditions	Applicability	Applicability	Applicability
Solution	Solution	Structure	Structure	Specification	Solution	Structure	Structure Participants	Structure
Benefits	Consequences	Consequences	Benefits		Benefits		Consequences	Consequences
Disadvantages			Limitations		Disadvantages			
Examples		Known Uses	Known Uses	Example	Application and Examples	Examples	Known Uses Sample Code	Known Uses Sample Code
Implementation					Examples	Implementation	Implementation	Implementation
Variations		Variations	Variations			Variations		
Related Patterns					Related Patterns		Related Patterns	Related Patterns

Table 3.4: Comparison of characteristics for design pattern description

3.3 Design Pattern Language for Model Transformations

In this section, we define the language we have created, DelTa, to express the solution characteristic of the unified template. DelTa is a neutral language, independent from any MTL. It is designed to define *design patterns* for model transformations, hence it is not a language to define model transformations. We could have used an existing MTL as a notation for DelTa; however, our need is a notation that expresses how elements within a rule are related and how rules are related with each other. In this respect, DelTa offers concepts borrowed from most MTLs, abstracts away concepts specific to a particular MTL, and adds concepts to more easily describe design *patterns*. This is analogous to how Gamma et al. [40] used UML class, sequence and state diagrams to define design patterns for object-oriented languages. In the following, we describe the abstract syntax, concrete syntax, and informal semantics of DelTa. We also compare DelTa with existing similar-purpose languages.

3.3.1 Abstract Syntax

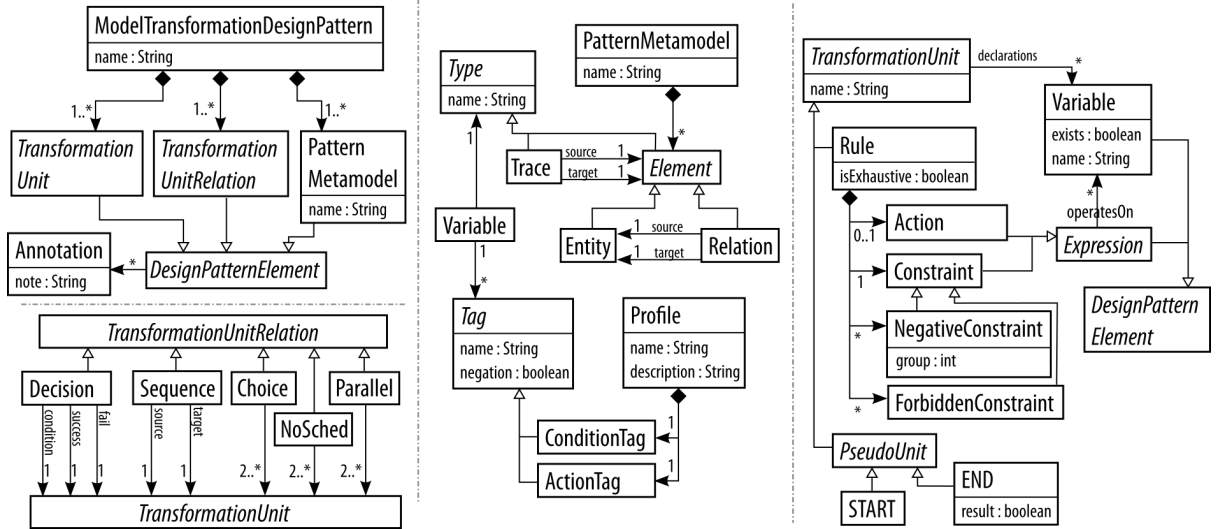


Figure 3.1: DelTa Metamodel

As depicted in Figure 3.1, a MTDP consists of three kinds of components: Transformation Unit (TU), Pattern Metamodel (PM) and Transformation Unit Relation (TUR). This structure is consistent with the structure of common MTLs [96]. In MTDP, rules represent a similar concept to graph transformation rules [31]. A rule consists of a con-

straint, an action, optional negative constraints, and forbidden constraints. The first three correspond to the usual LHS, RHS and NACs in graph transformation, respectively. A constraint is the precondition of the rule. A negative constraint defines the pattern that shall not be present, and a forbidden constraint only has a symbolic meaning that specifically says the elements shall not exist in the concrete transformation. Elements belong to a specific negative constraint group when multiple negative constraints are needed. Other than these two, a regular constraint, which can also be considered as a positive constraint, defines the pattern that must be present in the model. The action defines the changes to be performed on the constraint (e.g., creation, deletion, or update).

PMs and variables form the participants collaborating in a design pattern. There are two types for variables: an element from the PM or a trace. The PM is a label to distinguish between elements from different metamodels, because a MTDP is independent from the source and target metamodels used by the concrete model transformation. When creating a model transformation from a MTDP, the pattern metamodel should not be confused with the original metamodel of the source and/or target models of a transformation, but ideally be implemented by their ramified version [64]. Given the metamodel of a modeling language, ramification produces two metamodels, one to be used as the type model of the pre-condition pattern of a transformation rule and another for the post-condition pattern. For example, the former is used to perform queries on the input model of the transformation and the latter is used to perform updates to produce the output model. Metamodel elements are abstracted to entities and relations. All variables are strongly typed. Tags are of two kinds: either a condition tag to be used in constraints or an action tag to be used in actions. When creating a model transformation from a MTDP, the use of tags may require to extend the original or ramified metamodels with additional attributes. Traceability links are crucial in MTLs but, depending on the language, they are either created implicitly or explicitly by a rule. In DelTa, we opted for the latter, which is more general, in order to require the model engineer to take into account traceability links in the implementation.

As surveyed in [96], different MTLs have different flavors of TUs. For example, in

MoTif, an ARule applies a rule once, an FRule applies a rule on all matches found, and an SRule applies a rule recursively as long as there are matches. Another example is in Henshin [8] where rules with multi-node elements are applied on all matches found. Nevertheless, all MTLs offer at least a TU to apply a rule once or recursively as long as possible, where we adopt the latter with an *isExhaustive* attribute in the rule. All other flavors of TUs can be expressed in TURs as demonstrated in [96].

As surveyed in [23,98], in any MTL, rules are subject to a scheduling policy, whether it is implicit or explicit. For example, AGG [100] uses layers, MoTif and VMTS [73] use a control flow language, and GReAT [3] defines causality relations between rules. As shown in [97], it is sufficient to have mechanisms for sequencing, branching, and looping in order to support any scheduling offered by a MTL. This is covered by the five TURs of DelTa: Sequence, Choice, Parallel, Decision, and NoSched that are explained in Section 3.3.3. PseudoUnits mark the beginning and the end of the scheduling part of a design pattern.

Finally, annotations can be placed on any design pattern element in order to give more insight to the reader on the particular design pattern element.

3.3.2 Concrete Syntax

We provide both a graphical and a textual notation to represent DelTa instances, to satisfy the preferences of a wider spectrum of model engineers.

Graphical Concrete Syntax

We highlight the DelTa graphical concrete syntax through an example in Figure 3.2. The figure depicts a valid DelTa model although it is semantically not a design pattern.

1. A design pattern has a name and takes as a parameter the metamodels involved in the pattern. In this example, the **fixed-point iteration** design pattern involves one metamodel designated by **mm**.
2. A design pattern consists of a collection of rules rendered as rectangular blocks with their name appearing on the top-left. This pattern has five rules: **initiate**, **checkFixedPoint**, **Modify**, **Delete**, and **Create**. A concrete transformation rule

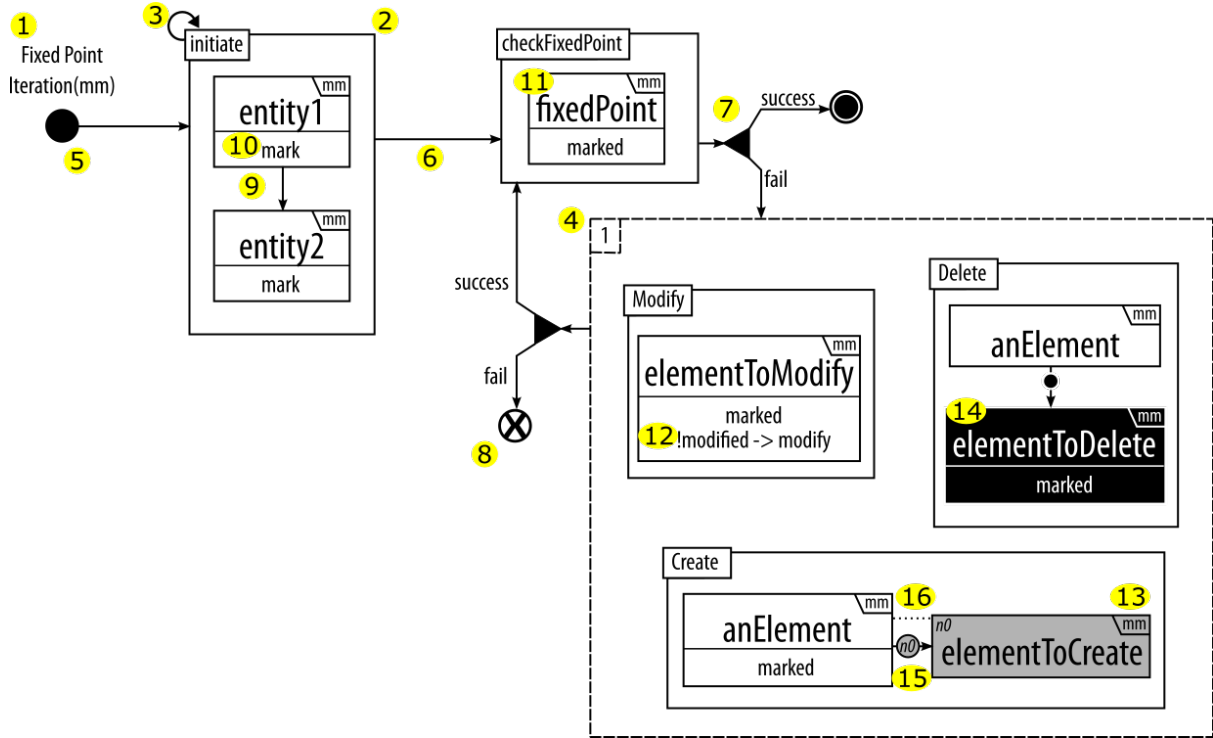


Figure 3.2: A Sample Pattern in DelTa Graphical Concrete Syntax

implementing this design pattern should have at least these rules.

3. When a self loop symbol appears on the top-left, the rule is set to be exhaustive. This means that the concrete transformation rule implementing it should be applied on all of its matches. This may require to have more than one rule implementing this rule, for example to match different metamodel types.
4. The dashed rectangle labeled “1” on the top-left represents a **choice** block. It states that at least one of the rules from this block should be implemented in the concrete transformation.
5. We use a control flow notation to represent rule scheduling. The start node (filled ball) indicates the initial rule of the design pattern.
6. Arrows between rule blocks indicate a precedence order: the concrete transformation rule implementing the **initiate** rule should be performed before the one implementing the **checkFixedPoint** rule.
7. Rule ordering may depend on the outcome of a rule. In this case, a decision node

determines the next rule based on whether a rule is successfully applied (matches are found) or not. For example, if a concrete transformation rule implementing the **checkFixedPoint** rule succeeds, the design pattern states that the transformation implementing it should terminate successfully (on a successful end node). Otherwise, the next rule to be applied should be from the choice block.

8. The design pattern can also state that the concrete transformation implementing it should terminate unsuccessfully. For example, if none of the concrete transformation rules implementing the rules within the choice block are applicable, then the design pattern indicates that the transformation is unsuccessful: in the design pattern, this means that a fixed-point is not reached.
9. DelTa rules have the minimal constraints and actions on elements of the metamodel that concrete transformation rules implementing them should have. For example, in rule **initiate**, there is only one constraint stating that there must be a relation from an entity (**entity1**) to another entity (**entity2**). Both entities shall belong to the same metamodel (**mm**). In DelTa, we only reason about entities and relations, independent from specific metamodel types and relations. Entities are represented using a UML class notation and their metamodel appears on the top-right.
10. Action tags, represented using UML attribute notation, indicate an action to be performed, by the concrete transformation rule implementing it, on the entity when stated in the imperative form. For example, **entity1** has the **mark** action tag, meaning that this entity must have been “marked” in some form at this step of the concrete transformation.
11. When stated as a past participle, it is a condition tag that the entity must satisfy in the constraint of the rule. For example, **fixedPoint** has the **marked** condition tag, meaning that this entity must have been “marked” in a previous rule so that a fixed-point is reached.
12. The notation $!modified \rightarrow modify$ should be interpreted as if the entity **element-ToModify** was not yet modified, then it should be modified after the application

of rule **Modify**.

13. Color coding of entities and relations inside the rules indicate whether they are part of the constraint or a type of action of the rule. White elements form the minimal application pre-condition that a concrete transformation rule implementing it should have. Gray elements are the minimal elements to be created in the concrete transformation rule. For example, the **Create** rule states that the concrete transformation rule implementing it should look for an entity that is marked and create a new entity **elementToCreate** and a relation to this entity.
14. Black elements are the minimal elements to be deleted in the concrete transformation rule. For example, the **Delete** rule states that the concrete transformation rule implementing it should look for an entity **elementToDelete** that is marked and is the target of a relation from another entity. Then, the rule should delete the entity **elementToDelete** and the relation.
15. Elements can also participate in the NAC of a DelTa rule. This is presented by labeling the element with the letter **n** followed by a number. A NAC indicates the pattern that should not be found by the concrete transformation rule implementing it. For example, the **Create** rule states that the concrete transformation rule implementing it should create the relation and the entity **elementToCreate** only if **elementToCreate** is not already connected to the marked entity **anElement**, because these two elements are annotated with **n0**.
16. Apart from entities and relations, **traces** are also types of elements that can be used in DelTa rules. They are represented as dashed lines between entities and/or relations. Just like other elements, they can be created and deleted, or be part of the constraint of a rule.

The complete description of the graphical concrete syntax is also available in Appendix B.

Listing 3.1: EBNF Grammar of DelTa in XText

```

1  MTDp:
2      'mtdp' NAME
3      'metamodels:' NAME (',' NAME) * ANNOTATION?
4      ('rule' NAME '*' ? ANNOTATION?
5          Entity?
6          Relation?
7          Trace?
8          Constraint
9          NegativeConstraint*
10         ForbiddenConstraint*
11         Action) +
12     TURelation+ ;
13
14 Entity: 'Entity' ELEMENTNAME (',' ELEMENTNAME) * ;
15 Relation: 'Relation' NAME '(' ELEMENTNAME ',' ELEMENTNAME ')'
16         (',' NAME '(' ELEMENTNAME ',' ELEMENTNAME ')') * ;
17 Trace: 'Trace' NAME '(' ELEMENTNAME (',' ELEMENTNAME) + ')'
18         (',' NAME '(' ELEMENTNAME (',' ELEMENTNAME) + ')') * ;
19 Constraint: 'constraint:' '~'? (ELEMENTNAME | NAME) (',' '~'? (ELEMENTNAME | NAME) ) *
20             ANNOTATION? ;
21 NegativeConstraint: 'negative constraint:' (ELEMENTNAME | NAME) (',' (ELEMENTNAME | NAME) ) *
22                     ANNOTATION? ;
23 ForbiddenConstraint: 'forbidden constraint:' (ELEMENTNAME | NAME) (',' (ELEMENTNAME | NAME) ) *
24                     ANNOTATION? ;
25 Action: ('action:' (~'? (ELEMENTNAME | NAME) (',' '~'? (ELEMENTNAME | NAME) ) * ) )
26         ANNOTATION? ;
27 TURelation: (TURTYPE ( 'START' | ( NAME ( '[' NAME '=' ( 'true' | 'false' ) ']' ) ? ) )
28             (',' ( 'END' | NAME) ( '[' NAME '=' ( 'true' | 'false' ) ']' ) ? ) + )
29             | Decision;
30 Decision: NAME '?' DecisionBlock ':' DecisionBlock;
31 DecisionBlock: ( 'END' | NAME) ( '[' ( 'END' | NAME) '=' ( 'true' | 'false' ) ']' ) ?
32             (',' ( 'END' | NAME) ( '[' ( 'END' | NAME) '=' ( 'true' | 'false' ) ']' ) ? ) * ;
33 terminal NAME: ('a'..'z' | 'A'..'Z') ( 'a'..'z' | 'A'..'Z' | '0'..'9' ) * ;
34 terminal ELEMENTNAME: NAME '.' NAME ( '[' NAME '=' ( 'true' | 'false' )
35             (',' NAME '=' ( 'true' | 'false' ) ) * ']' ) ? ;
36 terminal ANNOTATION: '#' (! '#') * '#' ;
37 terminal TURTYPE: ( 'Sequence' | 'Choice' | 'Parallel' | 'NoSched' ) NAME ? ':' ;

```

Textual Concrete Syntax

We have also designed a textual concrete syntax for DelTa. Listing 3.1 shows the EBNF grammar implemented in Xtext⁴. The structure of a DelTa design pattern is as follows. A new design pattern is declared using the *mtdp* keyword. This is followed by a list of metamodel names. The rules are defined thereafter. The ‘*’ next to the name of the rule indicates that the rule is exhaustive. A rule always starts with the declaration of all the variables it will use in its constraints and actions. Then, the *constraint* pattern is constructed by enumerating the variables that constitute its elements. Elements can be prefixed with ‘~’ to indicate their non-existence. Flags are defined on elements within square brackets. Optional negative constraints can be constructed, followed by an action. The final component of an MTDp is the mandatory TUR definitions. A TUR is defined

⁴<http://www.eclipse.org/Xtext/>

by its type and followed by a list of rule names. Annotations are enclosed within ‘#’. Listing 3.2 show the textual equivalent of the sample DelTa instance in Figure 3.2.

Listing 3.2: A Sample Pattern in DelTa Textual Concrete Syntax

```

mtdp FixedPointIteration
metamodels: mm
rule initiate*
  Entity mm.entity1, mm.entity2
  Relation r1(mm.entity1, mm.entity2)
  constraint: mm.entity1, mm.entity2, r1
  action: mm.entity1[mark=true], mm.entity2[mark=true]
rule checkFixedPoint
  Entity mm.fixedPoint
  constraint: mm.fixedPoint[mark=true]
rule Modify
  Entity mm.elementToModify
  constraint: mm.elementToModify[mark=true, modify=false]
  action: mm.elementToModify[modify=true]
rule Delete
  Entity mm.anElement, mm.elementToDelete
  Relation r1(mm.anElement, mm.elementToDelete)
  constraint: mm.anElement, mm.elementToDelete[mark=true], r1
  action: ~mm.elementToDelete, ~r1
rule Create
  Entity mm.anElement, mm.elementToCreate
  Relation r1(mm.anElement, mm.elementToCreate)
  Trace t1(mm.anElement, mm.elementToCreate)
  constraint: mm.anElement[mark=true]
  negative constraint: mm.elementToCreate, r1, t1
  action: mm.elementToCreate, r1, t1
Sequence: START, initiate, checkFixedPoint
Choice c1: Modify, Delete, Create
Success: checkFixedPoint, END
Fail: checkFixedPoint, c1
Success: c1, checkFixedPoint
Fail: c1, END[result=false]

```

3.3.3 Informal Semantics

The semantics of MTDP rules is borrowed from graph transformation rules [31], but adapted for patterns. Informally, a MTDP rule is applicable if its constraint can be matched without any negative constraints. If it is applicable, then the action must be performed. Conceptually, we can represent this by: $constraint \wedge \neg neg1 \wedge \neg neg2 \wedge \dots \rightarrow action$. Forbidden constraints remove ambiguity in the pattern and are not in this representation because they can be achieved either by ignoring them in the generation or adding them as a constraint to the model transformation language. The presence of a negated variable (i.e., with `exists=false`) in a constraint means that its corresponding element shall not be found. Because constraints are conjunctive, negated variables are also combined in a conjunctive way. Disjunctions can be expressed with multiple negative constraints. Actions follow the exact same semantics as the “modify” rules in

GrGen.NET [42]. Variables present in the action must be created or have their flags updated. A variable may be assigned tags to pass elements between rules. Negated variables in an action indicate the deletion of the corresponding element. Tags are used to either update some elements or reuse some elements in other rules. This is similar to pivot passing in MoTif [98] and GReAT [3], and parameter passing in Viatra2 [106]. A condition tag should be used as a verb in past tense form and an action tag should be used in imperative form. In case these forms are the same, we distinguish between them by adding the word “did” at the beginning of the condition tag, i.e., “set > didSet.”

MTDP rules are guidelines to the model engineer and are not meant to be executed. On one hand, the constraint (together with negative and forbidden constraints) of a rule should be interpreted as *maximal*: i.e., a concrete model transformation rule shall find at most as many matches as the MTDP rule it implements. On the other hand, the action of a rule should be interpreted as *minimal*: i.e., a concrete model transformation rule shall perform at least the modifications of the MTDP rule it implements. This means that more elements in the LHS or additional NACs may be present in the concrete model transformation rule and that it may perform more Create Read Update Delete (CRUD) operations. Furthermore, additional rules may be needed when implementing a MTDP for a specific application. Note that the absence of an action in a rule indicates that we do not care about the actions of the rule.

The scheduling of the TUs of a MTDP must always begin with a START node and may end with a number of END nodes. The Sequence has a source and a target defines the temporal order between two or more TUs regardless of their applicability. The Choice is a group of rules that defines the non-deterministic choice to apply one TU out of a set of TUs. The Parallel lets the rules inside to be applied in parallel. The Decision defines a conditional branching and applies the TUs in the success or fail branches according to the application of the rule in the condition. Note that the Decision TUR can be used to define loop structures. The last TUR is the NoSched, which means the scheduling of the rules contained in this TUR is not important, such as within a layer of rules in AGG.

The translation of DelTa models to concrete model transformations in specific MTLs

will give a more precise semantics by translation.

3.3.4 Limitations of DelTa

Instead of DelTa, a formal specification language such as in [70] can also be used, but at the price of the understandability and ease of implementability. DelTa is not for architectural patterns, anti-patterns, or higher-order transformation patterns because it focuses on micro-architectures. Nevertheless, the purpose of DelTa is not only for the definition of a pattern, but also to assist the model engineer during the design of model transformations, through automation. Finally, there are different model transformation approaches: imperatively (Kermeta [38]), rule-based (MoTif [98]), relational (QVT-R [83]), using term rewriting (Stratego [108]), template-based (Xpand [59]), or by-example [105]. We only focus on rule-based transformations.

3.4 Comparison of DelTa with Existing Languages to Express Design Patterns

We identified two existing studies in Chapter 2 that are comparable to DelTa in terms of representing model transformations agnostic from MTLs.

The RDs by Guerra et al. [45] represents a language that automatically produces the implementation of the transformation. In RD, rules focus on mappings rather than constraints and actions in DelTa. Therefore, there needs to be at least two metamodels involved in the transformation to map with each other. The execution flow of RD supports sequencing rules, branching in alternative paths based on a constraint or non-deterministically choosing to apply one rule. DelTa also provides these control flow constructs, in addition to *parallel*, to apply rules in parallel, and *noSched*, to mark the order of the rules as not important. RD is inspired from QVT-R and ETL and is therefore more easily implemented in these languages. However, DelTa is inspired from graph-based MTLs, making it implementable in any MTL.

TSPEC by Lano et al. [71] describes the formal structure of a design pattern in model transformations. The purpose of TSPEC is to formalize and define complete transformations, whereas the purpose of DelTa is to represent an abstraction of snippets of a trans-

formation. TSPEC uses mappings with constraints to represent rules in a transformation. In contrast, DelTa provides mechanisms to create different kinds of relations within rules, including element mappings from source language to target language. TSPEC provides an LMM to represent the languages on which the transformations operate upon, which is comparable to the *pattern metamodel* part of DelTa for precisely specifying constraints. In addition, DelTa has these features to help represent the design patterns: explicit decision structure to identify the result of a rule in terms of success and failure, choice and no scheduled structures when the order of the execution is not important. In conclusion, we can state that DelTa is designed intentionally from an engineering perspective, to help engineers understand and implement patterns, and to generate transformations from it, whereas TSPEC formalizes the effects of a transformation and is used to analyze them.

3.5 Summary of the Unified Template

In this chapter, we have precisely described the model transformation design pattern language: DelTa. A language by itself is not enough to represent design patterns. Therefore, we have also unified the existing templates in order to better report design patterns. We can now redefine existing design patterns using the unified template as well as define new ones.

CHAPTER 4

FIXED-POINT ITERATION PATTERN

The identification of a model transformation design pattern is a very tedious task. It requires one to analyze many solutions to a common problem, evaluate each of them looking at the trade-offs needed when using them, and generalize the most effective solutions as a single model that abstracts the problem domain. In Chapter 2, we analyzed existing model transformation design pattern studies in the literature in order to organize them. In this chapter, we illustrate how we identified a new design pattern based on solutions to the Lowest Common Ancestor (LCA) problem [4] using model transformations in different application domains. We present several iterations that gradually provide a better solution with respect to core metrics.

4.1 Running Example

LCA [4] is a general problem in graph theory where the task is to find the closest common ancestor between two nodes in a directed tree. Essentially, LCA attempts to find the lowest shared ancestor between two given input nodes of the tree. Although there are well-known solutions to this problem [24, 47, 87], we are interested in solutions implemented as model transformations. To this end, we assume the existence of a simple metamodel for trees with edges and labeled nodes. Figure 4.1 is a model instance of such a metamodel and the LCA of nodes D and G is node B.

4.1.1 Naïve Solution

Typically, solutions using model transformation approaches tend to take advantage of the declarativeness and non-determinism of rule-based systems. In the first solution presented, we first create all ancestor links of every node exhaustively as depicted by the first three rules in Figure 4.2. The first two FRules create an ancestor link to the

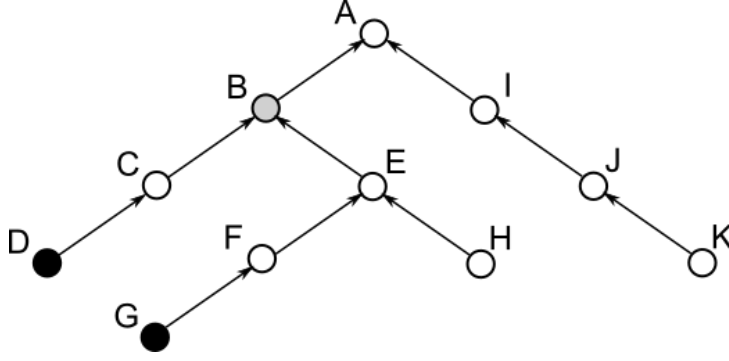


Figure 4.1: Tree instance for LCA problem

immediate parent of each node and to each node itself¹. The **LinkToAncestors** rule effectively computes the transitive closure between paths of connected nodes [2] because it is applied recursively since it is encapsulated in an SRule. After these three rules are applied, every node in the model has ancestor links, which are represented by AToMPM generic links: dashed arrows with a diamond in the center. Then, **GetLCA** rule marks the first common ancestor node of the given two initial nodes (A and B). To achieve that, we use the pivot feature in MoTif which forces the rule to parametrize these two nodes for further processing. The **GetLCA** rule also ensures with a NAC that the result (C) is the lowest common ancestor by preventing another ancestor node before the result (D) in the ancestor hierarchy.

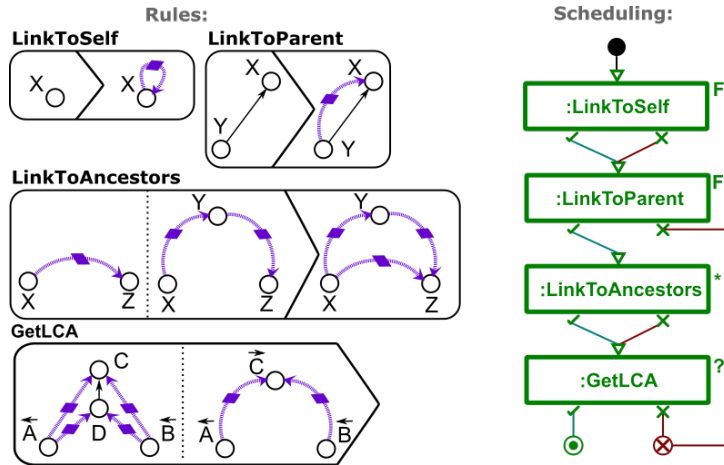


Figure 4.2: Rules for naïve solution

¹We create ancestor links to the nodes themselves. However, a node is not an ancestor of itself. The strategy is to find the correct solution if one of the 2 input nodes is the ancestor of the other.

Analysis

For this study, we focused on three metrics: *the number of rule applications* counts how many times the rule is applied, *the size of the rule* counts the number of elements present in the patterns of each rule, and *the number of auxiliary elements created* counts the number of ancestor links created to compute the LCA.

To compute the metrics, we consider a tree with n nodes and hence $n - 1$ edges. The **LinkToSelf** rule creates self-ancestor links for all nodes, to cover the trivial case, and is applied n times, once for every node in the tree. The **LinkToParent** rule creates ancestor links to the parents of each node and is applied $n - 1$ times, once per edge. The **LinkToAncestors** rule creates ancestor links to all ancestors of each node, recursively. Therefore, the number of ancestor links is proportional to the depth of each node. The following equation gives the total number of ancestor links that need to be created, where k_i is the depth level of node i .

$$\sum_{i=1}^n k_i - 2 = O(n^2)$$

After all ancestor links are created, the **GetLCA** rule is applied only once and returns the LCA of the given input nodes.

4.1.2 Improved Solution

We notice that, in the naive solution, there are more ancestor links that were created than optimally needed. Therefore, we propose another solution that uses locality starting from the input nodes. We adopt an iterative approach and start creating ancestor links one step at a time and, at each time, we check for a solution. The rules and scheduling are depicted in Figure 4.3. The **LinkToSelf** rule creates self-ancestor links for the given input nodes only and therefore is applied only twice. We use the pivot feature to apply the rules on pre-marked elements. That is, A and B are parametrized nodes bound to nodes from the input model at run-time. Then, the **LinkToParent** rule creates ancestor links to the parents of input nodes, which is applied twice. This results in an intermediate form of the tree instance, which may possibly solve the LCA task. Therefore, we apply the **GetLCA** rule and try to find the LCA at this level. If we can not find a solution, we

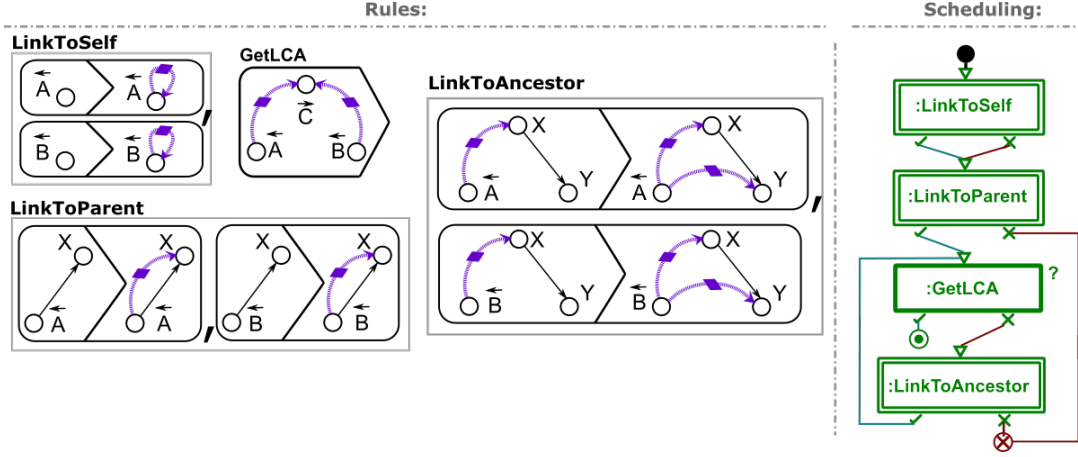


Figure 4.3: Rules for improved solution

Table 4.1: Metrics for naïve and improved LCA solutions

Rules	Size of rules		# Rule Applications		# Auxiliary Elements	
	<i>Naïve</i>	<i>Improved</i>	<i>Naïve</i>	<i>Improved</i>	<i>Naïve</i>	<i>Improved</i>
LinkToSelf	3	3	n	2	$n - 1$	2
LinkToParent	7	7	$n - 1$	2	$O(n^2)$	$2n - 2$
LinkToAncestors	14	14	$O(n^2)$	$2n - 2$	$O(n^2)$	$2n - 2$
GetLCA	14	14	1	n	0	0
Total	38	38	$O(n^2 + 2n)$	$3n + 2$	$O(n^2 + 2n)$	$2n + 2$

execute the **LinkToAncestor** rule and create one more level of ancestor links by using only the given input nodes again. With only one more step, this rule takes the intermediate form closer to a solution. Then, we use the **GetLCA** rule to check again. These iterative steps continue until the **GetLCA** rule finds a solution or the **LinkToAncestor** rule fails by not making a contribution to the solution (i.e., if the root is reached and **GetLCA** fails). For the tree instance and input nodes D and G in Figure 4.1, the solution is found in three steps. Therefore, the **GetLCA** rule is applied four times and the **LinkToAncestor** rule is applied three times. In general, the given input nodes might be in different depth levels (k_1 and k_2 , respectively). The ancestor link creation continues up to the root node, so the maximum of depth levels is the number of iterations needed to find the solution. In the worst case, this depth can be n and we create $n - 1$ ancestor links. Therefore, the **LinkToAncestor** rule is applied a total of $2(n - 1)$ times for input nodes and the **GetLCA** rule is applied n times.

Metrics for both the naïve and improved solutions are depicted in Table 4.1. One can

clearly see the improvement by comparing the metric counts between the naïve solution and improved solution. Without changing the size of the rules (i.e., the number of elements in each rule), we could reduce the number of rule applications and the number of auxiliary elements created (i.e., the ancestor links). These three metrics are related to the efficiency quality criteria. Therefore, we can say the improved solution is more efficient than the naïve solution by focusing on the worst case time complexity. Transformation execution time is irrelevant here since it is proportional to the metrics in Table 4.1.

4.2 Similar Problems in Different Domains

We observe that the solution to the LCA problem can be applied to other transformations in other domains as well. In this section, we identify and solve two more problems from different domains that have similar model transformation solutions.

4.2.1 Equivalent Resistance

In electrical circuits, it is common to compute the equivalent resistance of the whole circuit. Finding the equivalent resistance in a series of connected resistors is an interesting problem that can be solved with a similar model transformation to the LCA problem. In this case, the transformation takes as input an electrical circuit model with resistors connected both in serial and parallel. The rules are depicted in Figure 4.4. The **IsFinished** rule looks for resistors set in serial or parallel in the circuit. If the rule cannot find any more serial or parallel resistors, it will return the single resistor as the equivalent resistance. The **CalculateUnitEquivalentResistance** rule calculates equivalent resistance for only a set of serial and/or parallel resistors and directs the control flow to the **IsFinished** rule displaying a loop behavior. In this solution, we make small contributions to the model in order to find the result and check at each step for it.

Figure 4.5 depicts an input model instance of an electrical circuit and its final result after the transformation is applied. All circuits are reduced to find the equivalent resistance of the whole circuit.

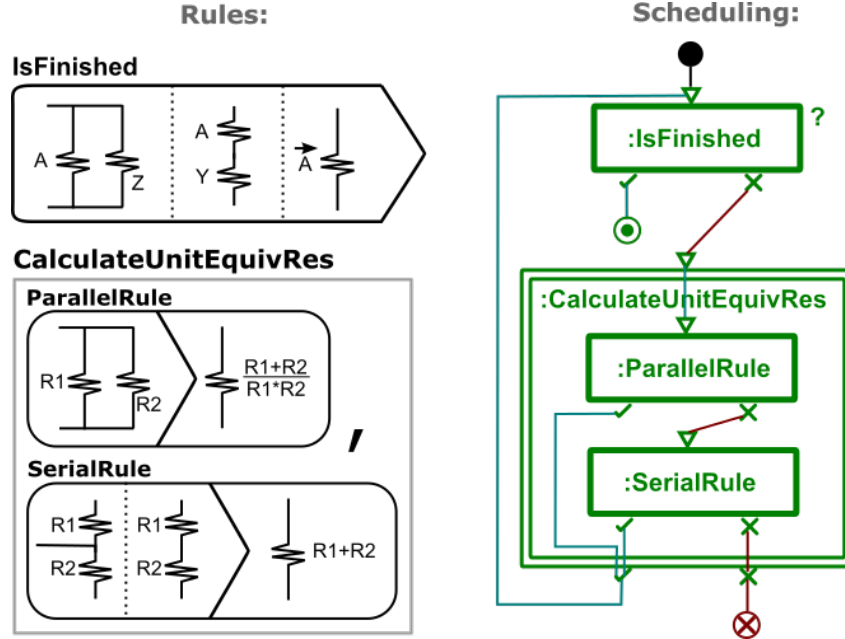


Figure 4.4: Rules for Equivalent Resistance Problem

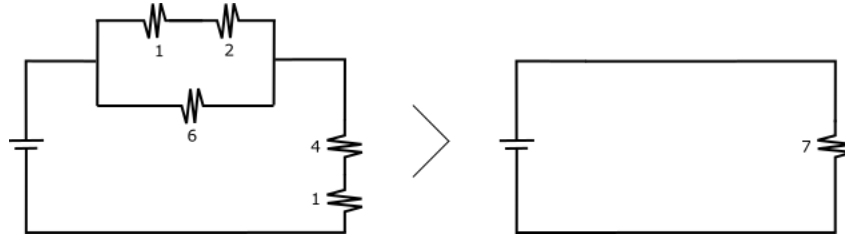


Figure 4.5: Sample input and output electrical circuits model

4.2.2 Dijkstra's Algorithm for Shortest Path

Dijkstra's algorithm is a well-known graph search algorithm that returns the shortest path and length of the path between two nodes, source and target [27]. The solution using model transformation² is provided in Figure 4.6 to find the shortest path from two input nodes, A and J. The input model is a directed and weighted tree. The `VisitImmediateNeighbors` rule initiates the algorithm by visiting the immediate neighbors of the source node. After a visit, each node is assigned with the weight of the path and is marked as visited (in red). The terminating criteria of the algorithm is whether all nodes have been visited, which is ensured by the `IsAllNodesVisited` rule. If there are still unvisited nodes, then the `VisitOneMoreHop` rule is executed. The `VisitOneMoreHop` rule selects the smallest number of weighted nodes among visited ones and calculates

²Dijkstra's algorithm may run better when implemented in a general-purpose language. The model transformation solution is for illustration purposes only.

the new weights for the unvisited neighbors of this node. After each node is visited, the target node will have the length of the shortest path as value and the path of arrows with a diamond in the middle will be the shortest path (i.e., the arrow in the `SelectLowest` rule RHS from node Z to node X).

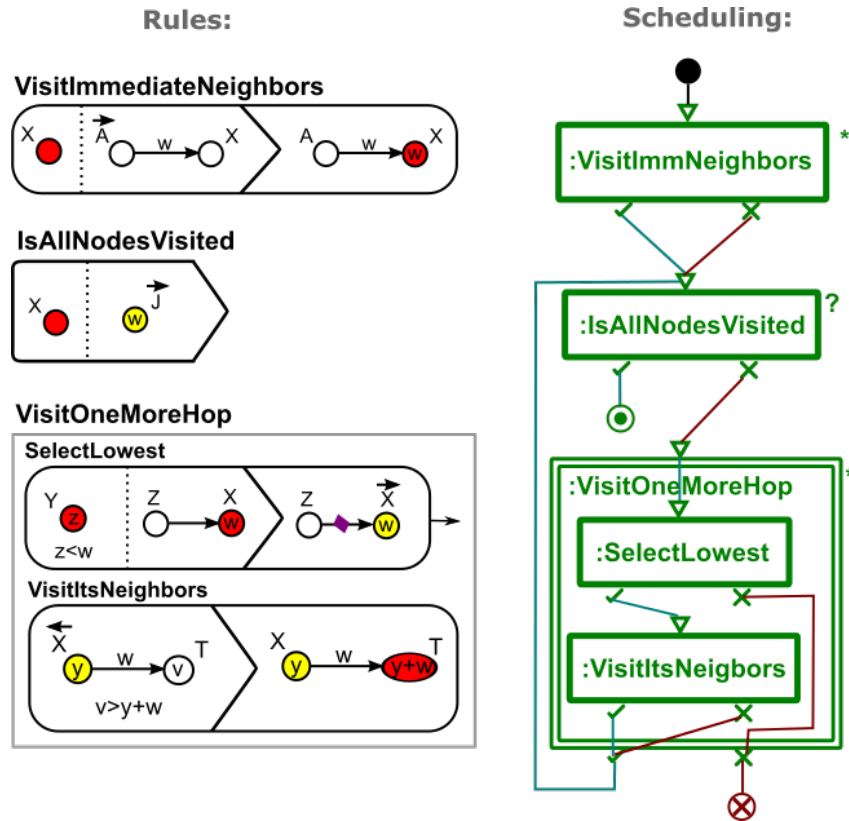


Figure 4.6: Rules for Dijkstra's Algorithm

Figure 4.7 depicts an input model instance of an directed graph and its final result after the Dijkstra transformation is applied. The shortest path from node A to J is computed and marked with purple diamond arrows.

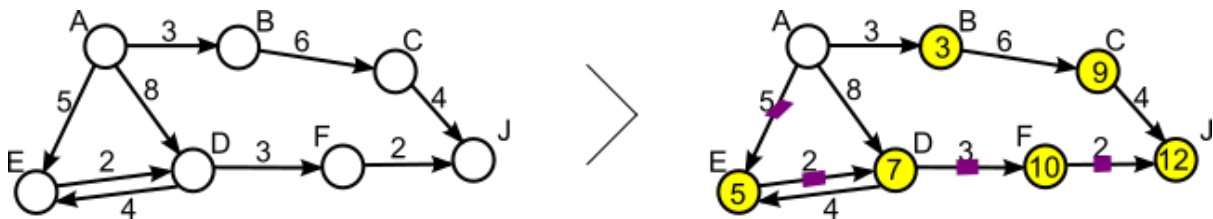


Figure 4.7: Sample input and output graph for Dijkstra transformation

4.3 Generalization of the Solution

The improved LCA, equivalent resistance, and Dijkstra’s shortest path model transformation solutions show similar characteristics. The overall strategy resembles that of a fixed-point iteration. In general, there are three blocks, as depicted in Figure 4.8. The first block initializes the input model with the creation of temporary elements and results in an intermediate form of the model (Initiate step). The initialization is optional (e.g., the Equivalent resistance problem did not need one). Then, a query verifies if a solution indicating the terminating criterion is found (Check step). Finally, if the query fails, the last block encodes how to increment the computation one step towards the final solution by manipulating the model with CRUD operations (Advance step). The structure can also be seen as a **while not** loop in programming languages.

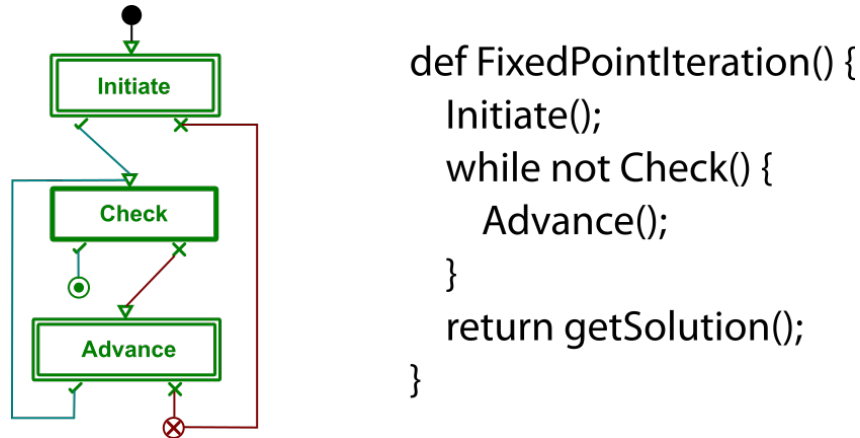


Figure 4.8: Generalization of the solutions with pseudocode

4.4 Promoting the General Solution to a Design Pattern

We adopt the unified template to describe the newly identified model transformation design pattern. The unified template provides better documentation and understanding for the pattern. It also helps the model engineer to implement the design pattern by following the structure depicted in DelTa.

Design Pattern: Fixed-point Iteration

- **Summary:** Fixed-point iteration is a pattern to represent a “do-until” loop struc-

ture. It solves the problem by modifying the input model iteratively until a condition is satisfied.

- **Application Conditions:** This pattern is applicable when the problem can be solved iteratively until a fixed point is reached. Each iteration must perform the same modifications on the model, possibly at different locations: either adding new elements, removing elements, or modifying attribute values.
- **Solution:** The solution is depicted in Figure 4.9. The pattern starts by marking a predetermined group of entities in the initiate rule and checks if the model has reached a fixed-point (i.e., the condition encoded in the constraint of the checkFixed-Point rule). If it has, the checkFixedPoint rule may perform some action, e.g., marking the elements that satisfied the condition. Otherwise, the pattern modifies the current model by choosing either create/modify/delete rules inside the Choice TUR. In this rule, only one of the rules in the block is selected and the fixed point is checked again after the application. If the rules in the block fail, it means no fixed-point is found and the result is a failure.

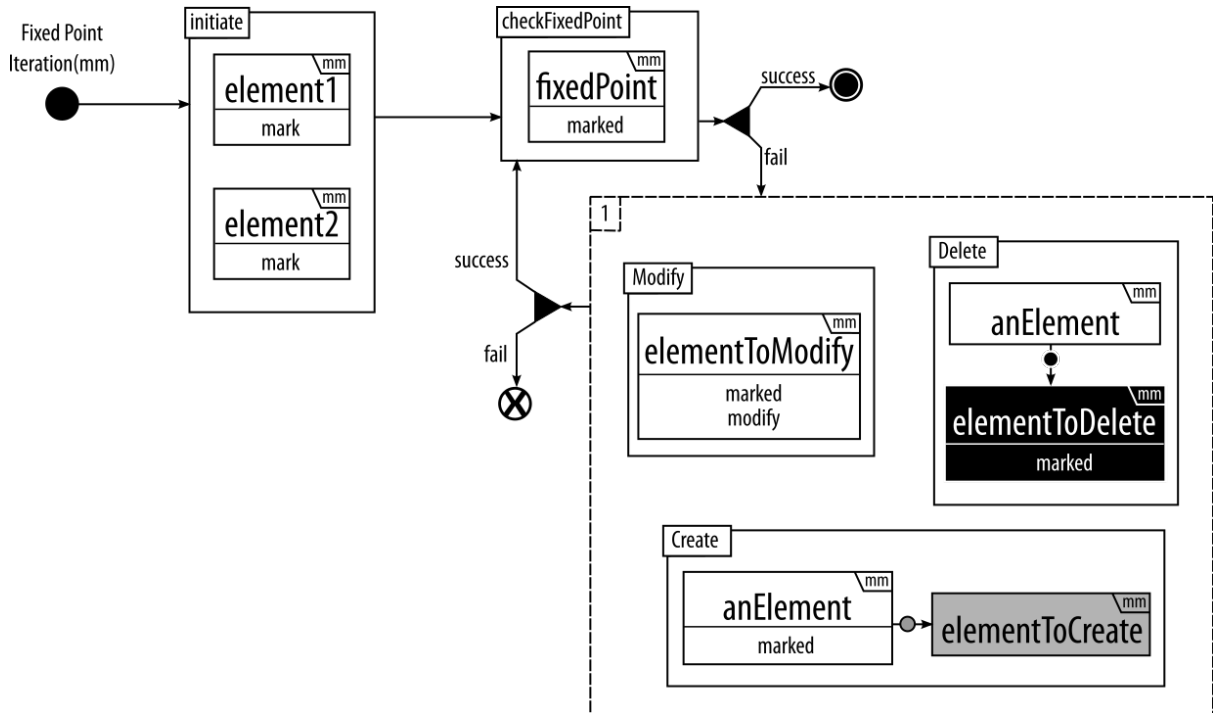


Figure 4.9: Fixed-point Iteration - Structure in DelTa

- **Benefits:** The pattern helps to traverse the graph structure of the input model. Therefore, it can be modified to fit into different graph traversal algorithms.
- **Disadvantages:** The traversal of the graph occurs iteratively, which hinders the parallelization opportunities of the model transformation.
- **Examples:** There are various applications of this pattern in different domains. In this chapter, we showed how to solve three problems with this pattern: computing the lowest-common ancestor of two nodes in a directed tree, finding the equivalent resistance in an electrical circuit, and finding the shortest path using Dijkstra's algorithm are some of them. Figure 4.3 shows the implementation of the LCA in MoTif using the fixed point iteration pattern. The `initiate` rule is extended to create traceability links on the input nodes themselves with the `LinkToSelf` rules and with their parents with the `LinkToParent` rules. The `GetLCA` rule implements the `checkFixedPoint` rule and tries to find the LCA of the two nodes in the resulting model following traceability links. This rule does not have a RHS but it sets a pivot to the result for further processing. The `LinkToAncestor` rule implements the `iterate` rule by connecting the input nodes to their ancestors. The MoTif control structure reflects exactly the same scheduling of the pattern.
- **Related patterns:** The iteration of the model with create/modify/delete elements can be done with the *phased construction* design pattern [71]. Also, auxiliary meta-model elements can be used in order to trace the elements.
- **Variations:** The pattern can be used to reduce the transformation by using delete-only rules, or augment the transformation by using create-only rules.

4.5 Summary of Identification of Design Patterns

In this chapter, we have solved a problem in two different ways and identified a model transformation design pattern by applying the solution to other problems in different domains that required a similar strategy to be solved. We have also analyzed the effect of applying this design pattern. Finally, we described the design pattern in the unified

template. Although this chapter shows only one design pattern, there are other design patterns we have identified in existing studies. All of them are presented in a catalog format in the next chapter in order to help model engineers when designing and implementing model transformations.

CHAPTER 5

MODEL TRANSFORMATION DESIGN PATTERN CATALOG

We believe that documenting the existing and newly identified design patterns is of crucial importance for model engineers in order to adopt them [40]. Therefore, in this chapter, we apply the unified template to the identified model transformation design patterns and propose a catalog. In the *implementation* field, where language-specific implementation details are provided, we illustrate each pattern with an example implemented in an actual MTL. The goal is to demonstrate the applicability of the unified template and represent the solution of the design patterns in DelTa. In the *related patterns* field, we provide the relation of the pattern with other patterns in the catalog if it exists. Furthermore, we specify the category under which each pattern falls according to the classification of Lano et al. [71].

5.1 New Model Transformation Design Patterns

This section covers two new model transformation design patterns. Both are identified by analyzing existing model transformation solutions, one of which was introduced in Chapter 4.

5.1.1 Fixed-point Iteration

This pattern falls under the “optimization” category and is described completely in Chapter 4.

5.1.2 Execution by Translation

This pattern falls under the “optimization” category.

- **Summary:** To execute a DSL, we often refer to some other languages that have well-defined semantics and easy to execute. This saves the time and effort of the

model engineer to write an executor from scratch for the DSL and standardizes the execution. With this pattern, the DSL is mapped to another intermediate language. Then, this language is simulated and the corresponding DSL elements are modified accordingly to show the animation.

- **Application Conditions:** The pattern is applicable when we want to execute a DSL using another DSL that has well-defined semantics.
- **Solution:** The pattern refers to two metamodels: the `dsl`, which is the DSL we want to execute, and the `simLang`, which is the intermediate language we simulate the `dsl`. Each element in the `dsl` is mapped to its corresponding equivalent in the `simLang` before the application of this pattern, using the Entities before relations pattern. In the `initialize` rule, we setup the initial state of the model ready for the simulation. The simulation runs in a loop. First, we check a `terminatingCondition` to know when to stop the execution. If it is not satisfied, the `simulate` rule is activated. In this rule, the state of specific elements needs to be modified according to a criterion in the `simulate` rule. Then, the `animate` rule finds the corresponding elements of the elements whose state has been modified in the `dsl` and does the necessary changes, which means either changing an attribute or the concrete syntax of those elements. Then, the `terminatingCondition` is checked again and the simulation continues.

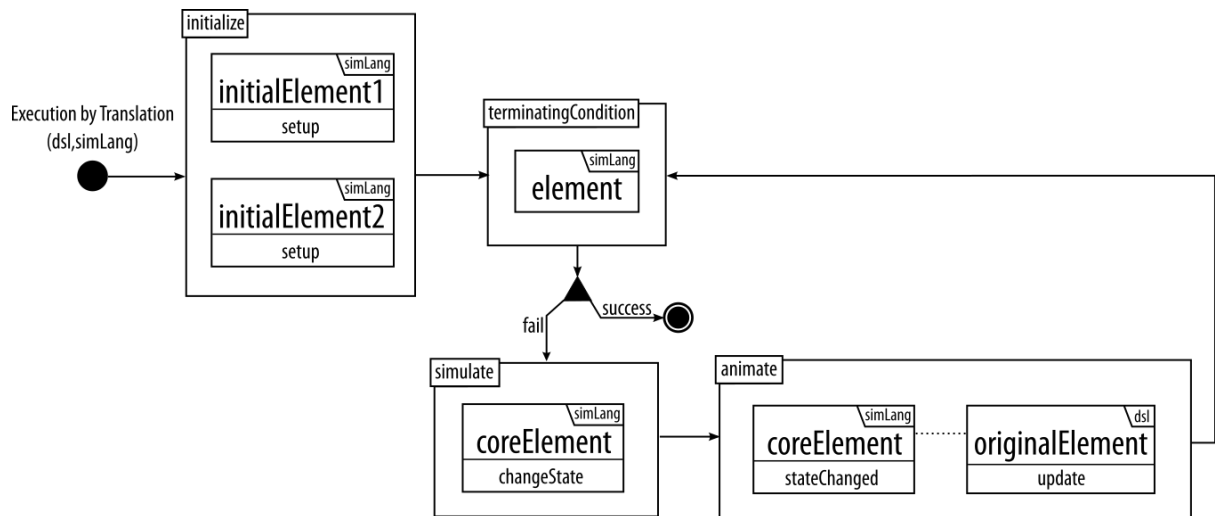


Figure 5.1: Execution by Translation - Structure in DelTa

- **Benefits:** The main benefit is not needing a separate execution driver for various DSLs. A well-known, well-analyzed executor can be reused for different DSLs.
- **Disadvantages:** The elements of the DSL should be mapped to the simulation language perfectly. Otherwise, there will be inconsistencies in the execution.
- **Examples:** In [64], Kühne et al. execute Finite State Automate (FSA) by translating to PNs. As they simulate the PN, they animate the FSA accordingly. In [94], we have defined a translation from an AD to a PN, and simulated the PN to animate the AD. De Lara and Vangheluwe mapped a production system DSL to a PN and used the PN for the dynamic behavior of the production system [25].
- **Implementation:** An implementation in MoTif is depicted in Figure 5.2. The example maps PNs to Statecharts (SCs) and uses the PN to simulate the SC.

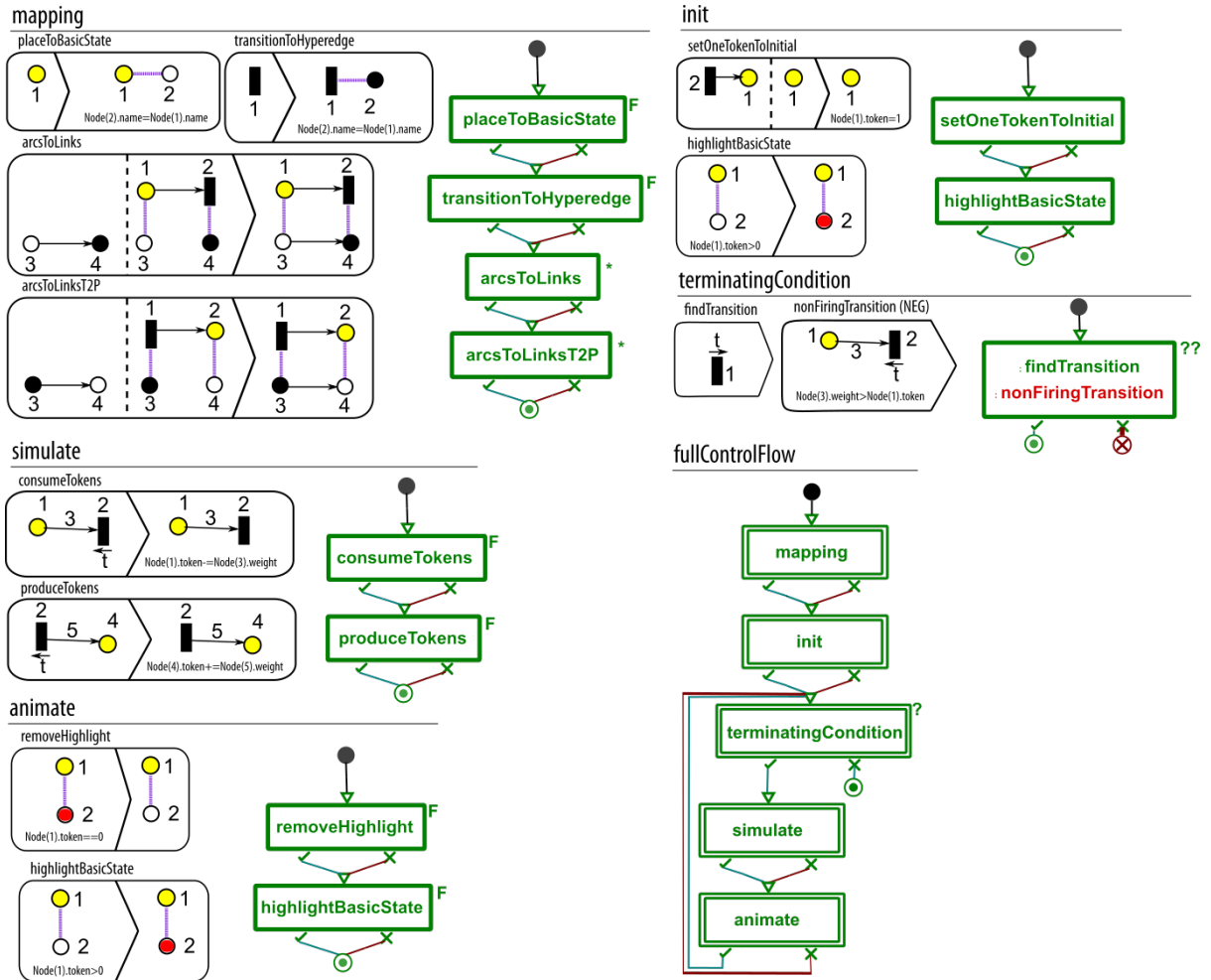


Figure 5.2: Petri Nets to Statecharts in MoTif

We only map the basic states and hyperedges in the SC for simplicity, but the advanced transformation can be found in [32]. The `mapping` part maps the places to basic states and transitions to hyperedges with the `placeToBasicState` and the `transitionToHyperedge` rules. Then, the arcs of the PN are mapped to links in the SC with the `arcsToLinks` and the `arcsToLinksT2P` rules. After mapping, the `init` part performs the same test as in the previous examples. The `setOneTokenToInitial` rule puts one token in the place of the initial node, which is the place without an incoming transition in this case. Then, the `highlight` rule highlights the current state. MoTif supports pivots, which is a built-in feature of the language to pass the matched elements between different rules. Therefore, this makes it easier to get a transition and check if it is firing or not by just passing it to the other rule, without the need for another attribute. A special complex query rule in MoTif makes it possible to get the firing transition with the help of the `findTransition` and the `nonFiringTransition` rules. The `findTransition` gets one transition, assigns a pivot to it and the `nonFiringTransition` checks if this transition is blocked or not. If the pattern is matched, that means it is not a firing transition and the rule tries another transition. The `simulate` and the `animate` part of the rules are the same as the previous examples, as they are basic PN simulation rules. In the `fullControlFlow` structure, one can realize that it looks similar to the structure of the “execution by translation” design pattern. This is because we borrow the control flow of DelTa, which is TUR, from the primitives of MoTif scheduling structures.

- **Related patterns:** Before application of this pattern, the elements of the `dsl` should be mapped to the elements of `simLang`. Therefore, this pattern should be preceded by a mapping pattern to perform the mapping of the source metamodel (e.g., `dsl`) elements to the target metamodel (e.g., `simLang`) elements.
- **Variations:** One variation is when the transformation simulates the first language and animates the second language accordingly. This only inverts the two metamodels in the four rules of this design pattern.

5.2 Generalized Model Transformation Design Patterns

This section covers the design patterns in existing studies but are generalized and redesigned in our unified template.

5.2.1 Entities Before Relations

This pattern falls under the “rule modularization” category.

- **Summary:** *Entities Before Relations* is one of the most commonly used transformation patterns in exogenous transformations to encode a mapping between two languages. It creates the elements in a language corresponding to elements from another language and establishes traceability links between the elements of source and target languages. This pattern was originally proposed in [52].
- **Application Conditions:** The Entities before relations pattern is applicable when we want to translate elements from one metamodel into elements from another metamodel.
- **Solution:** The structure of the pattern is depicted in Figure 5.3. The structure reads as follows. In the first rule, for each instance of entities in the source metamodel, if they do not have a corresponding target entity, create the corresponding entity in the target metamodel. The corresponding entity is represented by a trace connection between the source and target entities. Then in the second rule, relations are created between corresponding target entities, simulating their equivalent relations in the source metamodel, again if the relation does not exist. This ensures that first, all entities from the source are mapped to entities in the target and then,

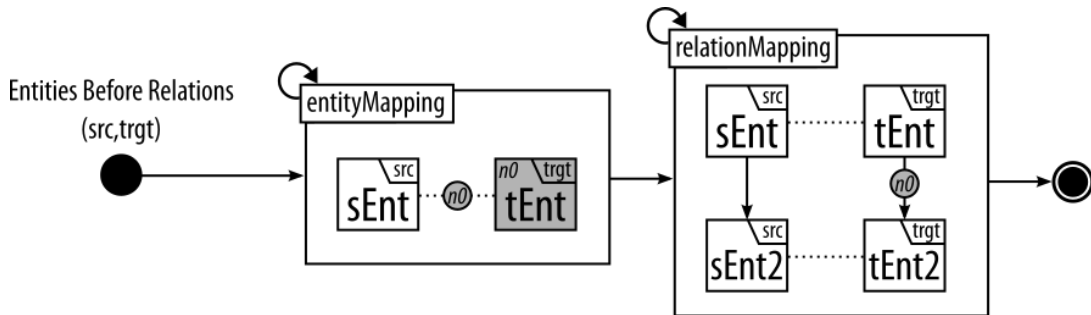


Figure 5.3: Entities before relations - Structure in DelTa

all relations between them are mapped.

- **Benefits:** With the help of traceability links, each element in the target language has a corresponding element in the source language. This improves debugging capabilities and error localization [54].
- **Disadvantages:** The pattern has no known disadvantages. However, the traceability links should be removed after the transformation is applied.
- **Examples:** A typical example of *Entities Before Relations* pattern is in the transformation from a class diagram to relational database diagrams, where, for example, a class is transformed to a table, an attribute to a column, and the relation between class and attribute to a relation between table and column.
- **Implementation:**

The implementation of the *Entities Before Relations* pattern in ATL is depicted in

```

module classdiagram2relationaldatabase;
create OUT : RelationalDB, trace : Trace from IN : ClassDiagram;

rule class2table {
  from
    sEnt : ClassDiagram!Class
  to
    tEnt : RelationalDB!Table ( name <- sEnt.name ),
    traceSandT : Trace!TraceLink ( source <- sEnt, target <- tEnt )
}

rule attribute2column {
  from
    sEnt2 : ClassDiagram!Attribute
  to
    tEnt2 : RelationalDB!Column ( name <- sEnt2.name ),
    traceS2andT2 : Trace!TraceLink ( source <- sEnt2, target <- tEnt2 )
}

rule attr2cols {
  from
    sEnt : ClassDiagram!Class ( attrs <- sEnt2 ),
    sEnt2 : ClassDiagram!Attribute,
    tEnt : RelationalDB!Table,
    tEnt2 : RelationalDB!Column,
    traceSandT : Trace!TraceLink,
    traceS2andT2 : Trace!TraceLink
  do {
    tEnt2.refSetValue('cols',tEnt2);
  }
}

```

Figure 5.4: Rules of Entities before relations pattern in ATL

Figure 5.4. It is applied to the class diagram to relational database transformation example. There are two rules that correspond to `entityMapping`: one for mapping classes to tables and one for mapping attributes to columns. The `relationMapping` is implemented as the `attrs2cols` rule. In ATL, traceability links are either implicit and created by the interpreter itself or modeled explicitly as a separate class connecting the source and target elements. We opted for the latter in this implementation. Due to the causality relation between the rules, this ATL transformation first applies rules `class2table` and `attribute2column`, then `attrs2cols` as stipulated in this design pattern.

- **Related patterns:** The pattern can be identified as a special case of *Phased Construction*, where the phases are, first, the entities and, then, the relations.
- **Variations:** The mapping can be done in either many-to-one or one-to-many with respect to the relation between source and target metamodels.

5.2.2 Visitor

This pattern falls under the “rule modularization” category.

- **Summary:** The *Visitor* pattern traverses all the nodes in a tree and processes each entity individually [36].
- **Application Conditions:** The *Visitor* pattern can be applied to problems that consist of (or can be mapped to) a tree structure where all the nodes need to be processed individually.
- **Solution:** The structure of the *Visitor* pattern is depicted in Figure 5.5. The *Visitor* pattern starts by marking an entity with an action tag in the *markInitEntity* rule. Then, in the *visitEntity* rule, the marked entity is tagged as processed, if it has not been processed yet. The *markNextEntity* rule marks the immediate child of the last processed entities as marked and returns back to the *visitEntity* rule. It accomplishes this with a decision relation and fail/success branches. The condition and action tags appear in the low compartment of the entity.

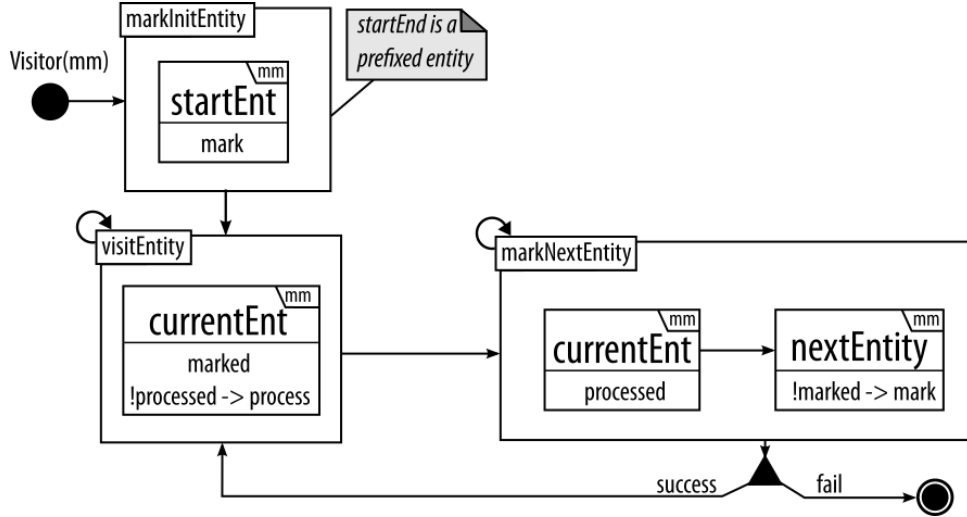


Figure 5.5: Visitor - Structure in DelTa

- **Benefits:** The pattern allows for the individual processing of nodes in a specific order, rather than bulk modification operations of model transformations. Note that a context can be provided when processing an entity of the metamodel. The pattern also allows for different model traversal strategies.
- **Disadvantages:** A loop helps to traverse the tree structure, therefore the parallelization of the rules is more difficult.
- **Examples:** This pattern can be used to compute the depth level of each class in a class inheritance hierarchy, which represents its distance from the base class.
- **Implementation:** Figure 5.6 depicts an implementation of the visitor pattern in GrGen.NET. This MTL provides a textual syntax for both rules and scheduling mechanisms. In a rule, the constraint is defined by declaring the elements of the pattern and conditions on attributes are checked with an if statement. Actions are written in a `modify` or `replace` statement for new node creation and `eval` statements are used for attribute manipulation. The `markBaseClass` rule selects a class with no superclass as the initial element to visit. Because this class already has a depth level of 0, we flag it as processed to prevent the `visitSubclass` rule from increasing its depth. This is a clear example of the minimality of a MTDP rule, where the implementation extends the rule according to the application. The `visitSubclass` rule

<pre> rule markBaseClass { e:Class; negative { d:Class; d-:subclass->e; } modify { eval { e.marked=true; e.processed=true; } } } </pre>	<pre> rule visitSubclass { d:Class; e:Class; d-:subclass->e; if { e.marked==true; e.processed==false; } modify { eval { e.processed=true; e.depth=d.depth+1; } } } </pre>	<pre> rule markSubclass { e:Class; f:Class; e-:subclass->f; if { e.processed==true; f.marked==false; } modify { eval { f.marked=true; } } } </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

```

exec markBaseClass
exec ([visitSubclass] ;> [markSubclass])*

```

Figure 5.6: Visitor rules and scheduling in GrGen.NET

processes the marked elements. Here, processing consists of increasing the depth of the subclass by one more than the depth of the superclass. The `markSubclass` rule marks subclasses of already marked classes. The scheduling of these GrGen.NET rules is depicted in the bottom of Figure 5.6. As stated in the design pattern structure, `markBaseClass` is executed only once. `visitSubclass` and `markSubclass` are sequenced with the `;>` symbol. The `*` indicates to execute this sequence as long as `markSubclass` rule succeeds. At the end, all classes should have their correct depth level set and all marked as processed. Note that in this implementation, `visitSubclass` will not be applied in the first iteration of the loop.

- **Related patterns:** The pattern is related to *Phased Construction* and *Recursive Descent* patterns [71], when the structure resembles a tree.
- **Variations:** The context that is needed to process elements can change. Also, `visitEntity` and `markNextEntity` rules can be NoSched rules with one rule per type inside to parallelize them. Finally, the ordering of the visit can be adapted to be depth-first, breadth-first, or custom order.

5.2.3 Transitive Closure

This pattern falls under the “rule modularization” category.

- **Summary:** The *Transitive Closure* is a pattern typically used for analyzing reachability related problems with an in-place transformation. It was proposed as a pattern in [2] and in [74]. It generates the intermediate paths between nodes that are not necessarily directly connected via traceability links.
- **Application Conditions:** The *Transitive Closure* pattern is applicable when the metamodels in the domain have a structure that can be considered as a directed tree.
- **Solution:** The solution is depicted in Figure 5.7. The pattern operates on a single metamodel. First, the `immediateRelation` rule creates a trace element between entities connected with a relation. It is applied recursively to cover all relations. Then, the `recursiveRelation` rule creates trace elements between the node indirectly connected. That is, if entities `child` and `parent` are connected with a trace, then `child` and `ancestor` will also be connected with a trace. It is also applied recursively to cover all nodes exhaustively.

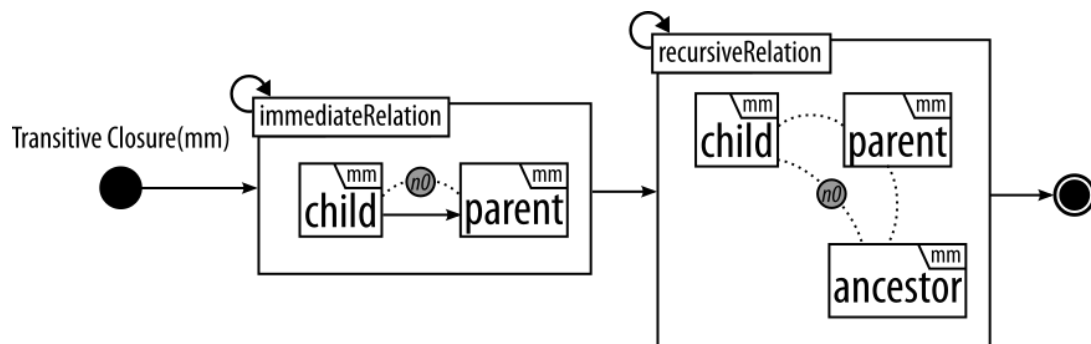


Figure 5.7: Transitive Closure - Structure in DelTa

- **Benefits:** Since all the trace elements are created from each element to all its ancestors, queries relying on information lookup are optimal. The resulting model is still valid conforming to its metamodel because trace links are created outside the scope of the metamodel. There are no side-effects and both rules are parallelizable.
- **Disadvantages:** The application of the pattern creates many trace elements for single elements that can create a memory overflow when the model is too large. We

need a rule for each type of relation, also for each combination of entity types, but that can be leveraged if using abstract types defined in the metamodel (i.e., super types can be used instead of the subtypes).

- **Examples:** The transitive closure pattern can be used to find the lowest common ancestor between two nodes in a directed tree, such as finding all superclasses of a class in a UML class diagram.
- **Implementation:** We have implemented the transitive closure in AGG. Figure 5.8 depicts the corresponding rules. AGG rules consist of the traditional LHS, RHS, and NACs. The LHS and NACs represent the constraint of the MTDP rule and the RHS encodes the action. The `immediateSuperclass` rule creates a traceability link between a class and its superclass. The NAC prevents this traceability link from being created again. The `recursiveSuperclass` rule creates the remaining traceability links between a class and higher level superclasses. AGG lets the model engineer assign layer numbers to each rule and starts to execute from layer zero until all layers are complete. Completion criteria for a layer is to execute all possible rules in that layer until none are applicable anymore. Therefore, we set the layer of `immediateSuperclass` to 0 and `recursiveSuperclass` to 1 as the design pattern structure stated these rules are to be applied in a sequence.

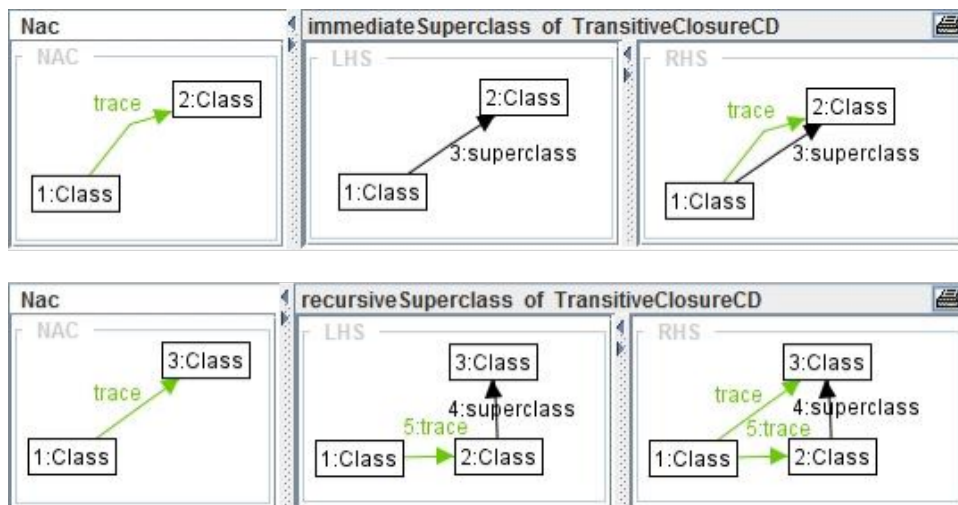


Figure 5.8: Transitive Closure rules in AGG

- **Related patterns:** *Transitive Closure* and *Fixed-Point Iteration* patterns can be integrated together to reach a target state in the model structure.
- **Variations:** Instead of traces, we can use existing relation types from the meta-model if allowed. Different types of relations can be used to provide a priority structure.

5.3 Lano et al.’s Model Transformation Design Patterns

In this section, we present the solutions of the existing design patterns by Lano et al. [71] using the unified template format. We only present the summary and solution fields because the rest of the description of the design patterns is already provided in their original paper.

5.3.1 Object Indexing

The behavior of this pattern is already used in previous patterns, because it is a built-in feature of DelTa.

- **Summary:** “All objects of an entity are indexed by a primary key value, to permit efficient lookup of objects by their key.” [71]
- **Solution:** The solution is depicted in Figure 5.9. In the “firstRule,” an entity is marked by setting a flag and in the “secondRule,” the same entity is used.

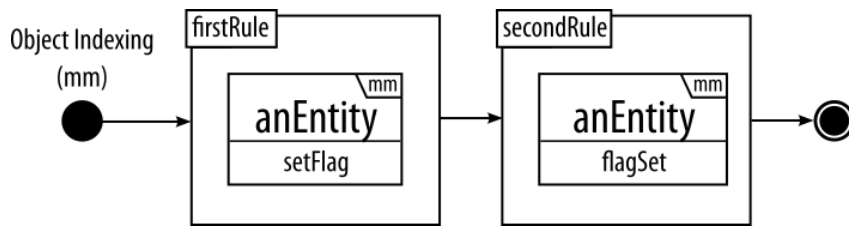


Figure 5.9: Object Indexing - Structure in DelTa

- **Variation:** Some MTLs provide internal mechanisms to support this design pattern (e.g., pivot structure in MoTif [98], GReAT [3] and VMTS [73]).

5.3.2 Top-down Phased Construction

- **Summary:** “This pattern decomposes a transformation into phases or stages, based on the target model composition structure. These phases can be carried out as separate subtransformations, composed sequentially.” [71]
- **Solution:** The solution is depicted in Figure 5.10. In the “formerPhase” rule, a container element “tContainer” of target metamodel is created and in the “latterPhase,” its composite element “tComposite” is created.

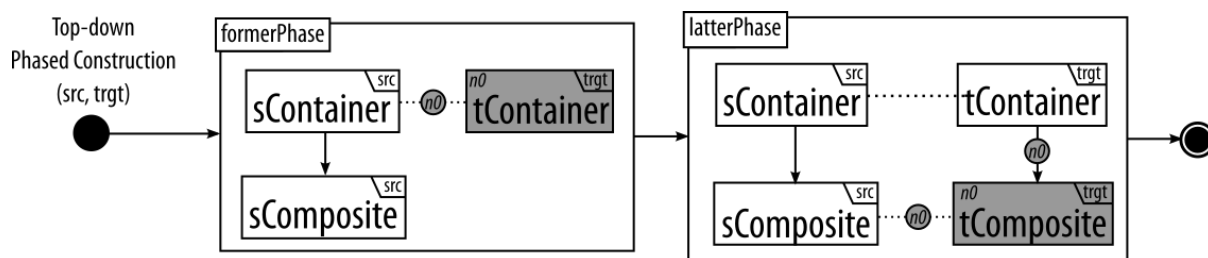


Figure 5.10: Top-down Phased Construction - Structure in DelTa

5.3.3 Parallel Composition

- **Summary:** This pattern separates the rules according to a distinguishable criteria in order to execute them in parallel, and elements of one parallel rule should not be accessed by another parallel rule in order to avoid conflicts.
- **Solution:** The solution is depicted in Figure 5.11. The “parallel1” and “parallel2” rules are to be executed in parallel and if “ent1” is updated in the first parallel rule, then it should not exist in “parallel2” rule, therefore it is marked with an “x” on top left in the latter rule. The same situation is true for “ent2” in the “parallel2” rule.

5.3.4 Unique Instantiation

- **Summary:** This pattern makes sure the created elements in a rule are unique and eliminates redundant creation of the same element by reuse.
- **Solution:** The solution is depicted in Figure 5.12. If “someEnt” element is created in a rule to be chosen from a group of rules, which are put inside a “NoSched”

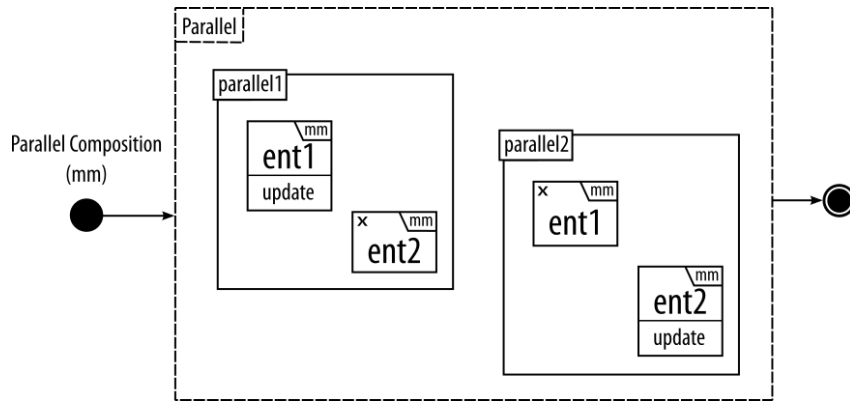


Figure 5.11: Parallel Composition - Structure in DelTa

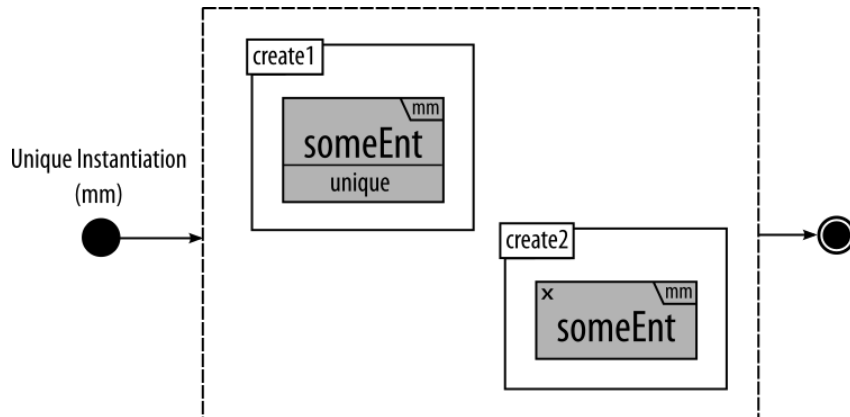


Figure 5.12: Unique Instantiation - Structure in DelTa

TUR, then it should not be created in another rule, which violates “someEnt”s being unique.

5.3.5 Entity Splitting

- **Summary:** This pattern separates the rules into pieces so that all creations must be done in its own rule when different types of target elements are created by the same source element.
- **Solution:** The solution is depicted in Figure 5.13. In the solution, “sEnt” is creating two different target elements, “tEnt1” and “tEnt2.” Therefore, they should be created in different rules grouped in a “NoSched” TUR.

5.3.6 Entity Merging

- **Summary:** This pattern separates the rules if the same target element is updated by different source elements. Each update by a different source element should

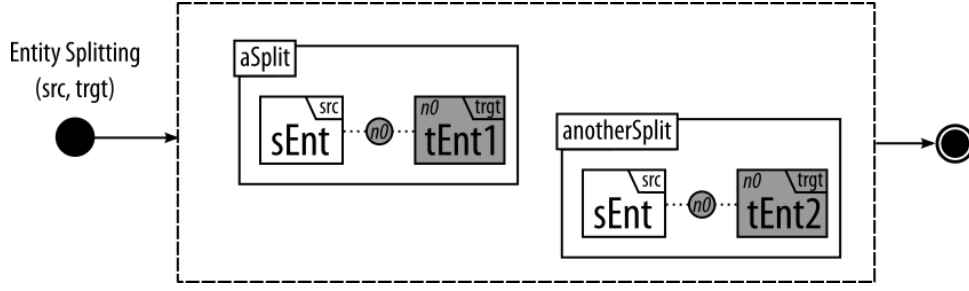


Figure 5.13: Entity Splitting - Structure in DelTa

occur within its separate rule.

- **Solution:** The solution is depicted in Figure 5.14. In the solution, after “tEnt” is created in the first rule, then it is updated by several different elements in the second NoSched TUR. Each update coming from different source elements should be in different rules.

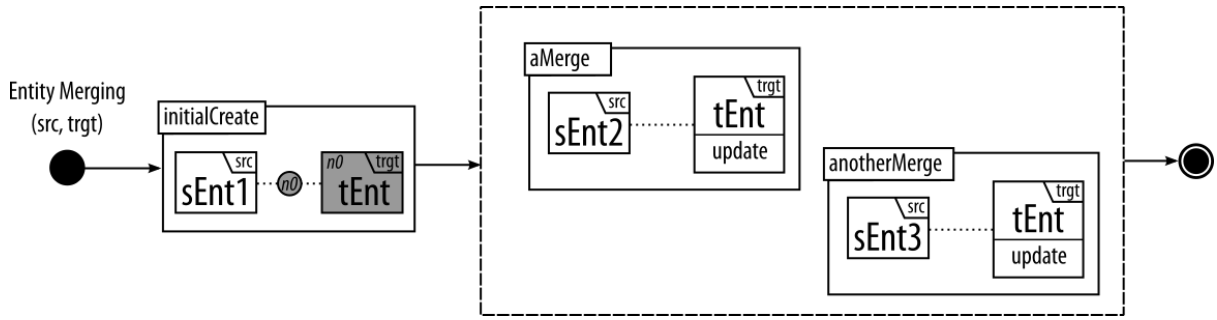


Figure 5.14: Entity Merging - Structure in DelTa

5.3.7 Construction & Cleanup

- **Summary:** “This pattern structures a transformation by separating rules which construct model elements from those which delete elements.” [71]
- **Solution:** The solution is depicted in Figure 5.15. The first set of rules only create the elements before the second set of rules, which only remove the elements. In the group, scheduling is not important. Therefore, rules are put inside a “NoSched” TUR.

5.3.8 Auxiliary Metamodel

- **Summary:** This pattern proposes to create an auxiliary metamodel for temporary elements used in the transformation that do not belong to either source or target

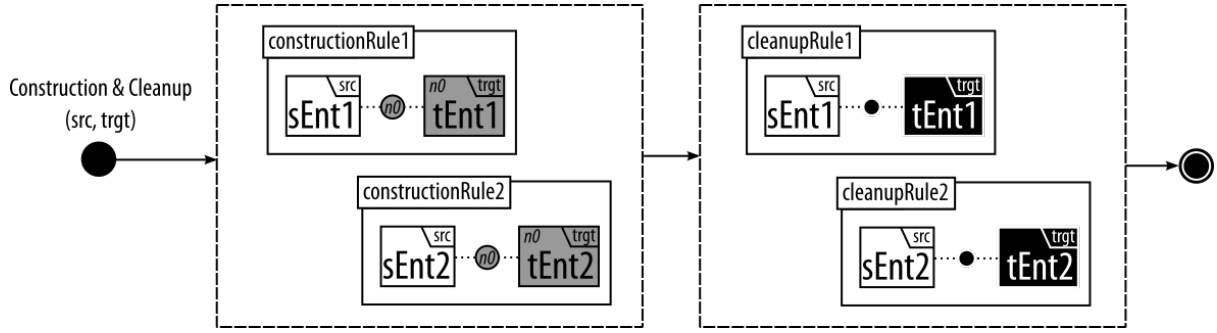


Figure 5.15: Construction & Cleanup - Structure in DelTa

metamodels.

- **Solution:** The solution is depicted in Figure 5.16. If any of the create, update, delete operations will be applied to the target metamodel entities, the same or similar operation should also be applied to their corresponding auxiliary metamodel elements i.e., “aEnt1,” “aEnt2,” and “aEnt3.” These auxiliary elements can be traced from either the source element or the target element.

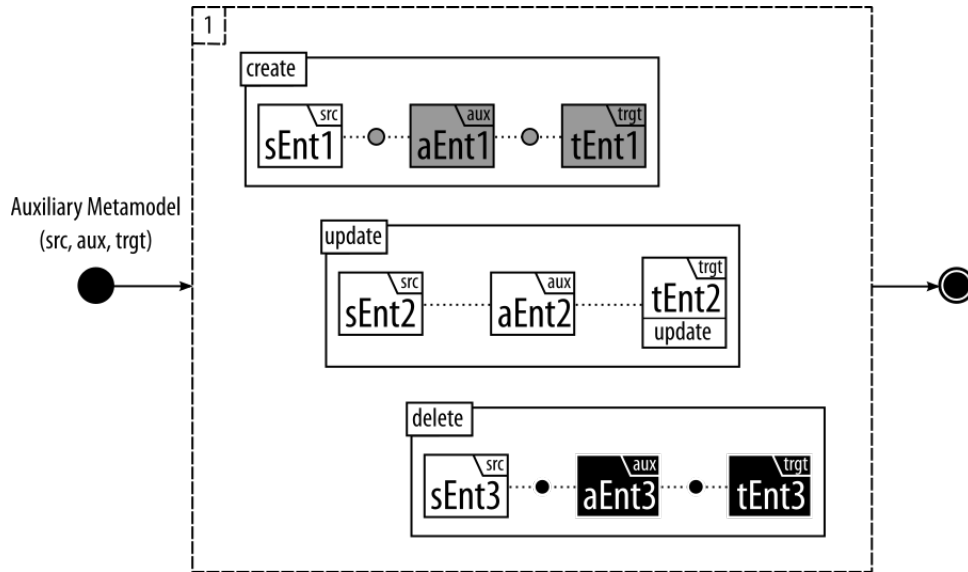


Figure 5.16: Auxiliary Metamodel - Structure in DelTa

5.3.9 Simulating Explicit Rule Scheduling

- **Summary:** This pattern suggests “use of additional application conditions of rules to enforce relative orders of rule execution.” [71]
- **Solution:** The solution is depicted in Figure 5.17. In order to specify an ordering

between two rules in a MTL that does not have an explicit rule scheduling structure, the pre-condition of the “secondRule” requires that the post-condition of the “firstRule” is satisfied. The “firstRule” satisfies a constraint that can either be setting a flag or changing a property in a specific entity, that is chosen to control the simulation of the explicit rule scheduling. Then, the “secondRule” checks the same entity whether the same constraint is satisfied. This way we ensure that the “firstRule” is executed before the “secondRule.” Other scenarios can be designed easily, such as involving three rules or simulating a decision.

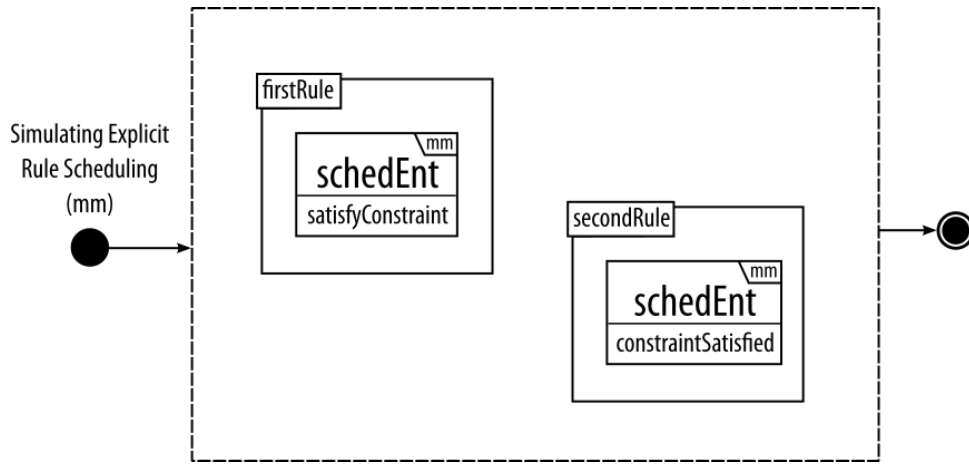


Figure 5.17: Simulating Explicit Rule Scheduling - Structure in DelTa

5.3.10 Simulating Universal Quantification

- **Summary:** The pattern simulates an antecedent “forAll(x|P)” condition by a double negation “not(X|not(P)).”
- **Solution:** The solution is depicted in Figure 5.18. In the solution, we intend to select some entities with a specific condition. However, graph transformation is existential. Therefore, we rewrite our rule using this pattern. Finally, the “select” rule tries to select entities that do not satisfy the condition and returns true if it can not find such a rule, and vice versa.
- **Implementation:** In the “terminatingCondition” rules of Figure 5.2, we show how this pattern is applied. In these rules, we want to select a firing “transition,” which means finding a “transition” with all incoming edges have token weights either equal

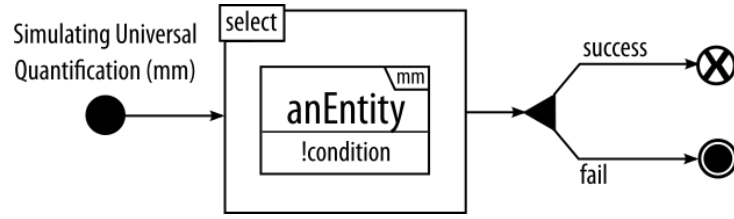


Figure 5.18: Simulating Universal Quantification - Structure in DelTa

to or less than tokens of their corresponding “places.” We rewrite the rules using this pattern and try to select a firing “transition,” if and only if that “transition” does not have the negative condition of a firing “transition,” which has less token weight in the incoming edge than tokens of its corresponding “place.”

5.4 Summary of the Catalog

In this chapter, we provided a list of model transformation design patterns in the form of a catalog to help model engineers. The catalog has explained each pattern in detail with solutions in DelTa and examples. A catalog is helpful in terms of documenting the design patterns. However, there needs to be a strategy about how to incorporate the existing design patterns while developing model transformation solutions. Adopting the design patterns from the very beginning will help create better quality model transformations.

CHAPTER 6

DESIGN PATTERN DRIVEN DEVELOPMENT OF MODEL TRANSFORMATIONS

In this chapter, we propose a process based on design patterns for model engineers to follow when addressing a specific transformation problem. The process is adapted to the model transformation paradigm from Budinsky et al. [18], who applied it on the object-oriented paradigm. We support the process with a tool that automates the appropriate steps to help model engineers. In order to demonstrate the process, we rely on a running example described below.

6.1 Case Study: Petri Nets to Statecharts

The running example we use is the model transformation problem introduced in the Transformation Tool Contest 2013 [104]: converting PN models to SC models (PN2SC). PNs are mainly used to analyze functional and stochastic properties of software systems, while SCs are mainly used to contribute in the design of the system and generate code. Therefore, this transformation is very useful for software engineers bridging tool support for analysis, design and code generation. The challenge in the PN2SC problem is that PNs have a flat data flow structure, while SCs have a hierarchical control flow structure. The goal of PN2SC is to perform a syntactical transformation to SCs while preserving the semantics of the PN model. Note that, this case study is for illustration purposes of the methodology and does not completely preserve PNs semantics and only implements some of the PN execution traces in the equivalent SC. Figure 6.1 depicts an example PN model and its transformed SC equivalent.

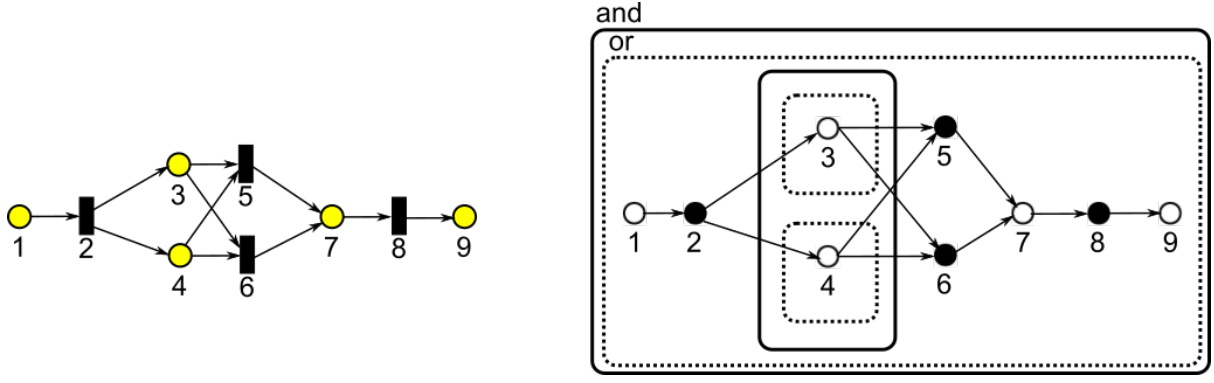


Figure 6.1: Sample Petri Nets model and its Statecharts equivalent

6.2 Design Pattern Driven Development Methodology

In the traditional way of solving a problem using model transformation, which is supported by the user study we conducted in Section 3.1, model engineers mostly start directly implementing the solution using their favorite model transformation language. In this section, we propose a methodology that will let them start with suitable model transformation design patterns and proceed with the implementation by customizing these patterns with respect to the problem at hand. This methodology promotes the use of model transformation design patterns at its core.

6.2.1 Problem Identification

The first step is to analyze the problem at hand and make sure that rule-based model transformation is the correct paradigm to solve the problem. The choice of the appropriate model transformation approach greatly influences the accidental complexities within the solution and thus the efficiency of the development [86]. A divide-and-conquer methodology has proven to be often useful to solve problems using model transformation, due to the modularity of the rules and control structure this paradigm offers [98]. Larger problems can be decomposed into simpler sub-problems, until a solution using a design pattern becomes apparent.

For the PN2SC problem, we chose a model transformation approach that uses an explicit scheduling of graph transformation rules, such as MoTif [98]. After analyzing the structure of each PN and SC, we pursue a bottom-up strategy by first creating a flat SC corresponding to the given PN model, keeping traceability links between PN places and

SC states created, as well as between PN transitions and SC hyperedges. Then, the idea is to incrementally remove PN elements while erecting an OR/AND-state structure in the SC. The next step is to create an AND-state for each set of places connecting to the same incoming and outgoing transitions. After recursively performing this step, we create an OR-state for every transition connecting exactly two places. After both reduction steps are complete, we end up with exactly one place that corresponds to the root state in the final SC. To design this transformation, we suggest addressing the following sub-problems in this order:

1. **Initialization rules:** design rules to map every PN element to its corresponding SC elements.
2. **AND Reduction rules:** design rules to perform the AND reduction.
3. **OR Reduction rules:** design rules to perform the OR reduction.
4. **Clean up rules:** design rules to clean up the remaining PN places.
5. **Rule scheduling:** design the overall control flow determining the partial order of execution of the rules.
6. **Simulation:** execute the resulting SC to verify that the trace output corresponds to the output by the original PN given one token in the root place.

6.2.2 Pattern Selection

For each transformation (sub-)problem, the model engineer selects a model transformation design pattern that is the best fit to solve it, as widely practiced in object-oriented design pattern selection [48,112]. This requires model engineers to scan through a design pattern catalog. For now, our catalog in Chapter 5 and other studies [2,35,52] constitute a corpus of 22 design patterns for model transformation, which is a good starting point. This is an important step, because correctly implementing an inappropriate design pattern will certainly lead to a bad design [17]. One should not assume there is a design pattern for every problem. However, if a design pattern can solve the problem at hand, then it is highly recommended to use it. If not, one possibility is to craft a solution in

DelTa. The DelTa model helps the model engineer focus on the design of the solution so that they are not encumbered with the details of an implementation in a specific MTL. This approach of DelTa is similar to the approach followed by transML [45].

For PN2SC, we identify the following patterns for each sub-transformation. In the initialization phase, elements are mapped by using the *Entities Before Relations* pattern. Some of these mappings can be performed in parallel using the *Parallel Composition* pattern. We use the *Auxiliary Metamodel* pattern to create trace links between PN and SC elements. SCs represent a hierarchical structure, therefore, we use the *Phased Construction* pattern to create AND and OR states in the reduction phases, i.e., create the AND (OR) states first, then the sub-states within those. The reduction will run for as long as the PN satisfies the pre-conditions for reduction: this recursion is performed using the *Fixed-point Iteration* pattern. Selecting the correct part of the PN for reductions requires the reuse of objects previously matched or created in other rules: for this we use the *Object Indexing* pattern. The overall simulation of the SC using the corresponding PN is ensured using the *Execution by Translation* pattern.

6.2.3 Adaptation to the Problem

Design patterns are described in a generic way to be independent from the specific context of their application. Thus, the model engineer must adapt the pattern to the problem at hand. This includes customizing the participants of the pattern in the DelTa model: e.g., metamodels, entities, and relations. The adaptation step can also include investigating variants of the pattern, focusing on the variation or related patterns fields.

In PN2SC, most of the previously identified design patterns use a PN as the source metamodel and an SC as the target metamodel. In addition, all the names in the patterns should be adapted to represent the appropriate PN and SC structures. There are some patterns that must be used in variation to satisfy the needs of the transformation. The *Phased Construction* pattern should be bottom-up [71] because we build the SC structure starting from the leaf nodes. In addition, in the initialization phase, we need a one-to-many mapping variant of the *Entities Before Relations* pattern, because an element in a PN is mapped to two elements in a SC. The structure of the pattern is depicted in

Figure 5.3. The pattern recommends mapping entities before relations.

The design pattern is generic and it does not represent a solution for a specific model transformation problem. Therefore, we have to customize this design pattern by setting `src` and `trgt` to the respective paths of the PN and SC metamodels. These settings continue with specifying what each entity/relation/trace type will be converted in the rules. Part of the customization of the *Entities Before Relations* pattern is listed in Table 6.2.3.

Element in DP	Equivalent in Transformation
<code>entityMapping</code>	<code>transitionToHyperedge</code>
<code>src</code>	<code>PetriNet</code>
<code>trgt</code>	<code>Statechart</code>
<code>sEnt</code>	<code>Transition</code>
<code>tEnt</code>	<code>Hyperedge</code>
<code>t1</code> (name of trace)	<code>GenericLink</code>
<code>relationMapping</code>	<code>arcsToLinks</code>
<code>sEnt2</code>	<code>Place</code>
<code>tEnt2</code>	<code>State</code>

Table 6.1: Customizing the participants of the design pattern

6.2.4 Implementation and Refinement

At this point, the model engineer first implements the customized pattern as-is from the previous step. The choice of the MTL may require more effort at this step. For instance, if the MTL does not support explicit control scheduling, the model engineer also has to adopt the *Simulating Explicit Rule Scheduling* design pattern. Nevertheless, this step may be automated by generating a model transformation excerpt that implements the pattern [5]. Then, as a generic solution to the problem, the implemented design pattern needs to be further refined to the specific problem. At the rule level, one can add more actions to perform or expand the constraint of rules. Another refinement may be to add further rules to deal with different types. More concrete examples can be found in [35].

Complete Transformation

For PN2SC, we chose the MoTif [98] language available in AToMPM [99]. Figure 6.2 shows the final implementation of the initialization rules that map the PN elements to SC elements according to the customizations made in Table 6.2.3.

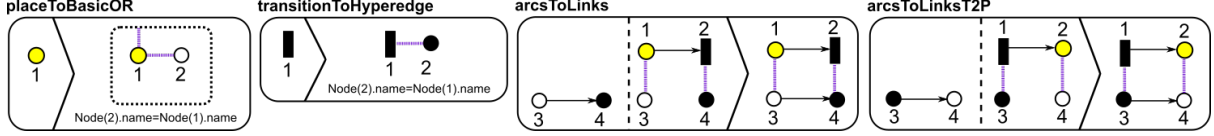


Figure 6.2: Initialization Rules

The rules for AND reduction are depicted in Figure 6.3. These rules match the AND state scenarios in the input PN model. These scenarios are: 1) all incoming places have same incoming and outgoing transitions, and 2) all outgoing places have same incoming and outgoing transitions. Then, an AND state is created for each match.

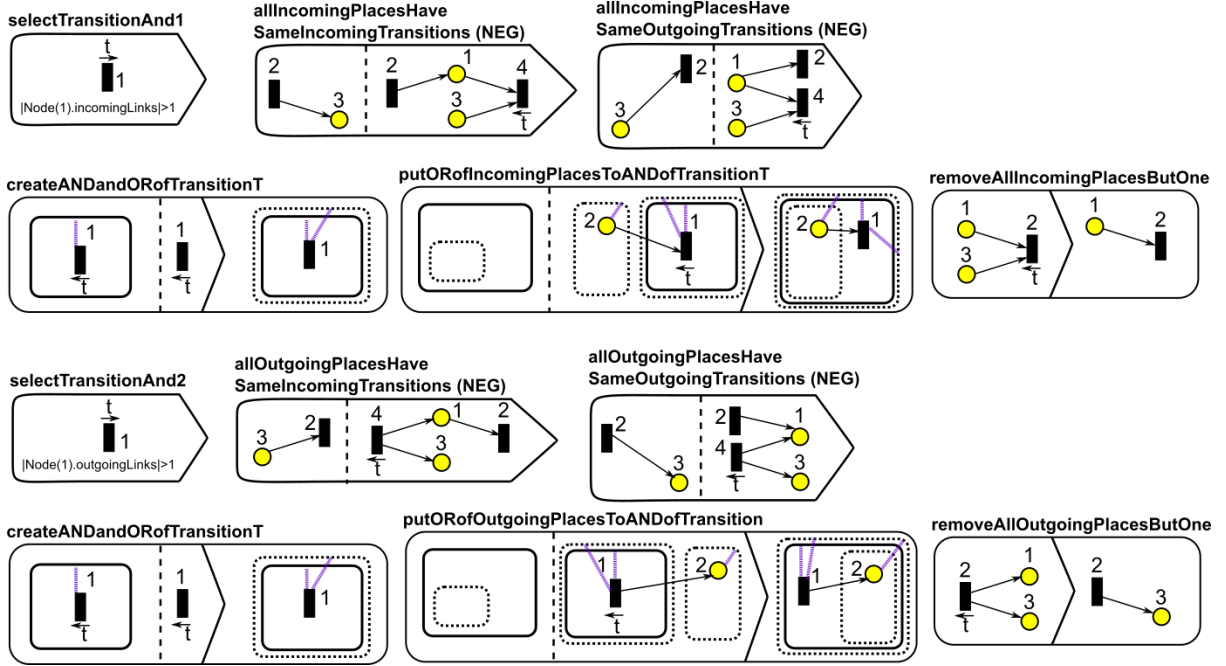


Figure 6.3: AND Reduction Rules

The rules for OR reduction are depicted in Figure 6.4. These rules match the OR state scenario by finding transitions that have a single incoming place and a single outgoing place. Then, an OR state is created for each match.

The removal of the PN elements are done in the final steps of Figure 6.3 and Figure 6.4.

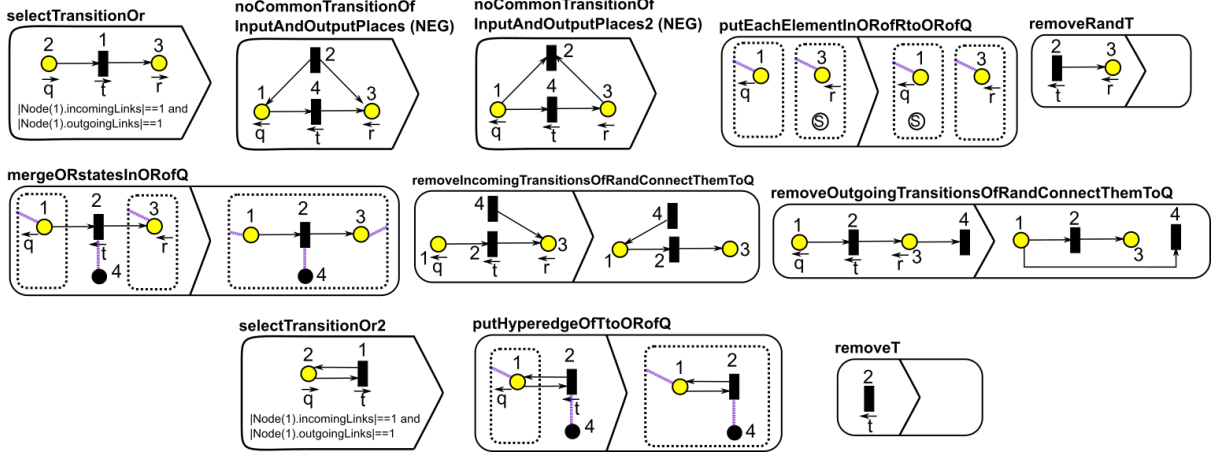


Figure 6.4: OR Reduction Rules

For the sake of simulating the SC at the end, we perform lazy deletion and mark the PN elements as deleted in order not to lose the traceability links between PN and SC elements. The final step creates the container SC and puts the remaining OR states in it. The rules are depicted in Figure 6.5.

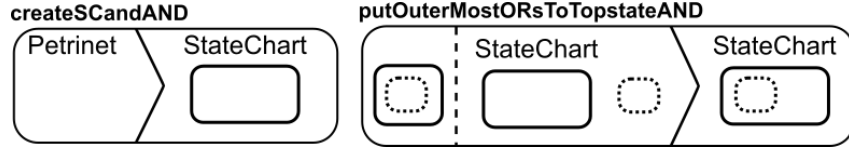


Figure 6.5: Finishing Rules

The overall scheduling of the rules is depicted in Figure 6.6. The figure shows the scheduling of the subproblems we have identified. The scheduling of the individual rules within these subproblems are available in [32].

Finally, we have simulated the SC model by simulating the underlying PN model.

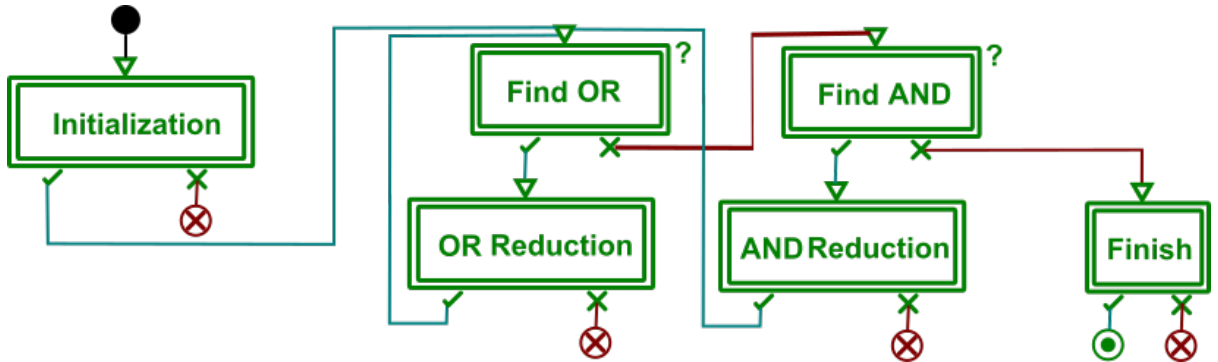


Figure 6.6: Scheduling of PN2SC Subproblems

Figure 6.7 depicts the rules and the scheduling for simulation. The PN is simulated based on firing transitions. A transition is fired only when all the incoming places have at least one token. When a transition is fired, the tokens of incoming places are consumed (i.e., removed) and the tokens of outgoing places are produced (i.e., increased by one).

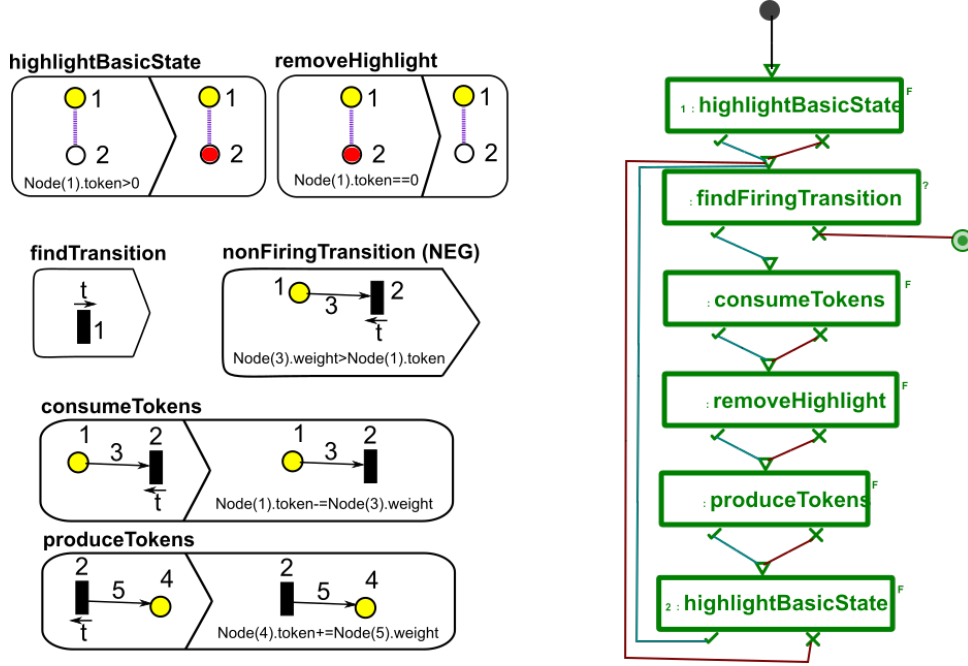


Figure 6.7: Simulation of SC using PN

6.2.5 Integration

The implemented patterns should be integrated carefully with the rest of the model transformation. Further customizations or modification may be required. In addition, micro-architectures can also be constructed by applying patterns in combination with each other [12].

6.2.6 Beyond the Process

This process is iterative and incremental since it can be repeated as long as sub-problems can be solved using design patterns. It can also be integrated in the transition between the design and implementation phases of well-known software development processes that must be followed in the project (e.g., Unified Process [14] or Agile Method [76]). This process does not assume there is necessarily a design pattern to solve the (sub-)problem at hand.

6.3 Automating the Methodology

We investigate automation opportunities in our methodology. The *Problem identification* step can be automated since it requires processing and interpreting a specific problem described in natural language. However, it is not a part of this dissertation. In the *Pattern selection* step, a set of design patterns can be proposed given a number of keywords or meta-data about the problem parts. More accurate data connected to the problem's structure will make the proposed design patterns more relevant. The *Adaptation to problem* step can be automated with the help of an additional metamodel to specify customizable fields of each pattern. The *Implementation and refinement* step is the main task that we automate in this section. This step is automated given the right generator for a specific model transformation language. The *Integration* step must be done manually since it is a tedious task and requires processing the existing system in detail. However, with some guidance from the model engineer, this step can utilize a semi-automatic guided process.

The automation of all steps are future research opportunities. In this section, we focus on the *Adaptation to problem* and *Implementation and refinement* and generate a partial and extendable model transformation solution from a design pattern in a concrete model transformation language using two approaches. The generated model transformation has the details needed for the transformation to be compiled successfully. However, the rest of the transformation logic with respect to the actual problem is left to the developer: this is the *Integration* step.

We propose two different approaches to produce a model transformation that are each fit for different types of MTLs. For both approaches, we rely on an additional model, DelTaConfig, that holds transformation-specific parameters, such as name or type of the elements. The DelTaConfig metamodel consists of key-value pairs that are similar to Table 6.2.3. An instance of it has to be created for each concrete transformation problem.

6.3.1 Code Generation with Xpand

The first approach to synthesize a transformation is to use template-based code generation. This approach is best fit for MTLs that have a textual concrete syntax or

not explicitly modeled. We implemented the solution in XPand¹. In this approach, we generate executable code directly from DelTa models. For this example, we choose GrGen.NET [42] as the target language. A transformation in GrGen.NET consists of a text file for rules and another one for their scheduling. Figure 6.8 depicts a basic workflow of the method.

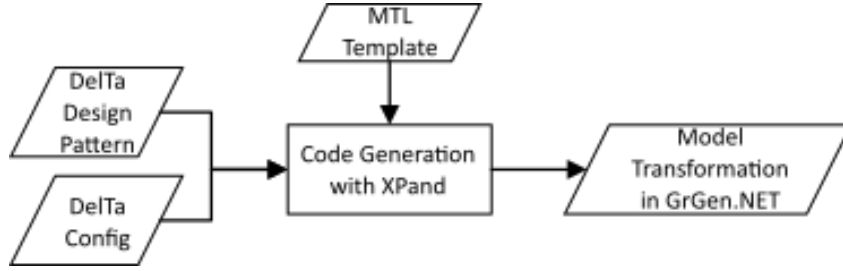


Figure 6.8: Code Generation Workflow

In order to generate code, we have created templates where each element in DelTa is mapped to its corresponding structures in GrGen.NET. An excerpt of these templates is depicted in Listing 6.1. Rules are straight-forward to generate. However, for the scheduling, we have to traverse the scheduling structures of DelTa recursively in order to find the correct application ordering of the rules starting from the `start` node. The parameters of the transformation are read from the DelTaConfig model encoded in the `getName` function.

We produce two files for GrGen.NET: one for the rules and one for the scheduling. The generated model transformation is checked for syntax conformance by running the GrGen.NET compiler. A snippet from the generated transformation and the scheduling is depicted in Listing 6.2.

6.3.2 Higher-order Transformation with ATL

For the second approach, we create a HOT [101]. This approach is best fit for MTLs that have been explicitly modeled, in particular in Ecore. Since the DelTa metamodel is already implemented in Ecore (as was required for the first approach), we chose a target MTL that also has a metamodel in Ecore, such as Henshin [8]. We implemented the HOT in ATL [54], a transformation language specializing in exogenous transformations.

¹<https://eclipse.org/modeling/m2t/?project=Xpand>

Listing 6.1: XPand Template for GrGen.NET

```

<<DEFINE main FOR ModelTransformationDesignPattern->>
  <<FILE getDPName( ) + '_RULES.grg'-->>
    <<EXPAND patternMetamodel FOREACH patternMetamodels->>
      using Trace;
    <<EXPAND unit FOREACH units->>
  <<ENDFILE->>
<<ENDDEFINE->>

<<DEFINE unit FOR Rule->>
  rule <<getName(this.name)>> {
    <<EXPAND constr FOR constraints->>
    <<EXPAND action FOR actions->>
  }
<<ENDDEFINE->>

<<DEFINE constr FOR Constraint->>
  <<EXPAND variable FOREACH operatesOn->>
<<ENDDEFINE->>

<<DEFINE action FOR Action->>
  modify {
    <<EXPAND variable FOREACH operatesOn->>
  }
<<ENDDEFINE->>

<<DEFINE variable FOR Variable->>
  <<EXPAND typeDef FOR this.type->>
<<ENDDEFINE->>

<<DEFINE typeDef FOR Trace->>
  <<this.source.name>>:<<getName('trace')>>-><<this.target.name>>;
<<ENDDEFINE->>

<<DEFINE typeDef FOR Entity->>
  <<this.name->>:<<getName(this.name)>>;
<<ENDDEFINE->>

<<DEFINE typeDef FOR Relation->>
  <<this.source.name>>:<<getName(this.name)>>-><<this.target.name>>;
<<ENDDEFINE->>

```

Listing 6.2: Transformation Snippet in GrGen.NET

```

using PN;
using SC;
using Trace;
rule transitionToHyperedge {
  sEnt:Transition;
  negative {
    tEnt:Hyperedge;
    sEnt--:trace->tEnt;
  }
  modify {
    tEnt:Hyperedge;
    sEnt--:trace->tEnt;
  }
}

-----
transitionToHyperedge * ;> arcsToLinks *

```

The input is a design pattern in DelTa along with a configuration model in DelTaConfig. The output is a Henshin model transformation. The workflow of the HOT is depicted in Figure 6.9.

The ATL transformation imports the DelTa, DelTaConfig, and Henshin metamodel. Figure 6.10 depicts a snippet from the HOT that transforms a DelTa rule into a Henshin rule. In order to set the node types of Henshin transformation elements in the HOT in ATL, we use reflection properties of ATL. We produce an Ecore model conforming to the Henshin metamodel. The generated model transformation is checked for syntax conformance by loading it with the Henshin editor. Figure 6.11 depicts an excerpt of the generated transformation.

6.3.3 Discussion of the Automation Approaches

We have explored two alternatives on how to generate concrete transformations from design patterns defined in DelTa. With the code generation approach, the generated concrete transformation can be further edited as needed directly. As for the HOT, the resulting model represents the abstract syntax of the transformation. In the case of Henshin, the language does not support editing the model with a graphical syntax if it is generated automatically. Nevertheless, the transformation can be further edited using the tree editor as depicted in Figure 6.11 or directly in XML Metadata Interchange (XMI), but this is a tool issue.

The HOT with ATL is outplace, meaning there is no option to modify the input transformation. Therefore, we also investigate using a HOT that allows for inplace transformations, such as Henshin. However, Henshin can only take one input model, the DelTa

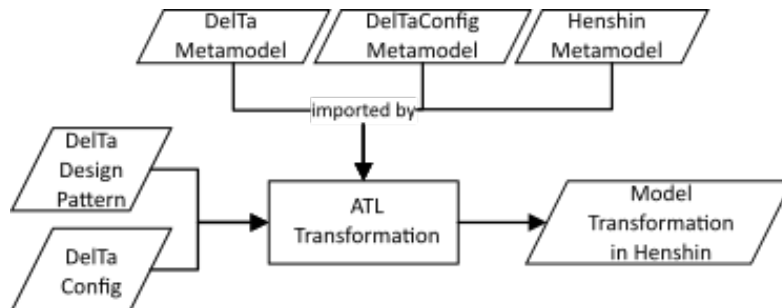


Figure 6.9: Higher-order Transformation ATL Workflow

```

rule DeltaRule2HenshinRule {
  from
    r: DelTa!Rule
  to
    hr: Henshin!Rule (
      name <- r.getMappedName(r.name)
    )
}

helper context DelTa!TransformationUnit def :
  getSequencedRules() :
    OrderedSet(DelTa!TransformationUnit) =
  if self.sourceBack.target.ocllsTypeOf(DelTa!Rule) then
    OrderedSet{}.append(self.sourceBack.target)
    .append(self.sourceBack.target.getSequencedRules())
  else
    OrderedSet{}
  endif
;

rule createSequence {
  from
    st: DelTa!Start
  to
    su: Henshin!SequentialUnit (
      subUnits <- st.getSequencedRules()
    )
}

```

Figure 6.10: ATL HOT Snippet for Henshin

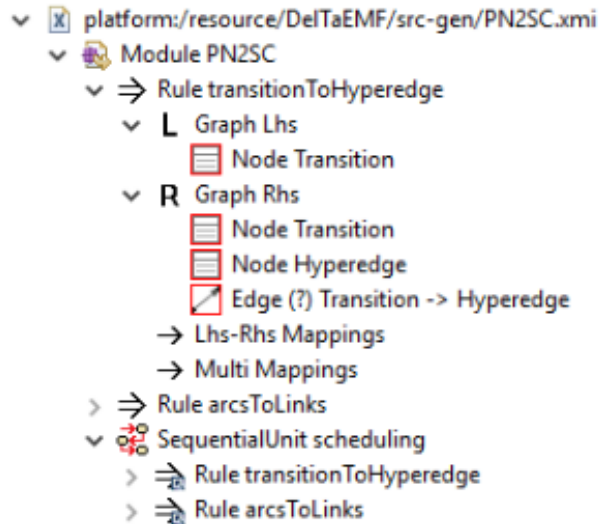


Figure 6.11: Transformation Snippet in Henshin

design pattern and it generates the draft of the transformation without the customizations made to the design pattern in order to adapt it to the problem. To apply the customization with DelTaConfig, we needed to resort to a second and redundant pass

using Extensible Stylesheet Language Transformations (XSLT) to modify the values of the parameters in the output transformation draft, which makes the HOT option with Henshin impractical.

Regardless of the approach chosen, the result is a partial yet concrete model transformation that is ready to be integrated into the rest of the transformation solving the problem at hand.

6.4 Tool Support to Guide Model Engineers

In order to alleviate the development process of model transformation for model engineers, we have implemented a tool that automates many forementioned steps of the process described when a design pattern is appropriate to solve the problem. The workflow of our transformation prototype is outlined by the activity diagram in Figure 6.12 and screenshot in Figure 6.13. The model engineer starts by selecting a design pattern; in this case the *Entities Before Relations* pattern. Each design pattern requires a DelTa-Config model of customization with problem-specific parameters to be filled out by the model engineer (e.g., the right-most area in Figure 6.13 shows parameters that need to be filled for the problem). The model engineer then selects the target MTL; in this case GrGen.NET. The generator loads the template corresponding to the selected MTL, the DelTa model corresponding to the selected pattern, and the parameters to finally synthesize the concrete model transformation excerpt. The model engineer completes the transformation manually to solve his specific problem or can generate other excerpts of the transformation from other design patterns.

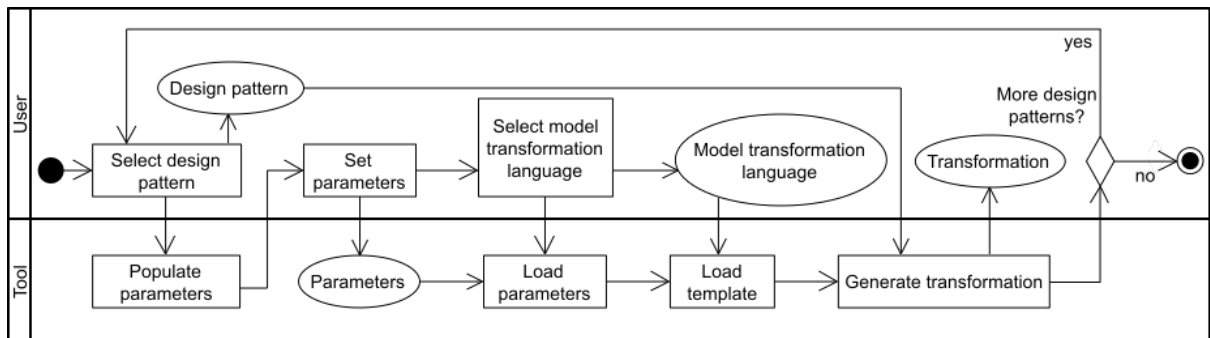


Figure 6.12: Activity diagram to generate a model transformation

We have implemented the prototype with a graphical user interface to simplify the process. The window depicted in Figure 6.13 helps the model engineer to select a design pattern: upon choosing a pattern from the list at the top left, the solution of the pattern in the DelTa graphical syntax appears below it. The model engineer can also read the complete description of the design pattern with all the fields by clicking on the **Show Design Pattern Details** button. Then, the model engineer fills problem-specific details about the design pattern, following Budinsky et al.'s [18] code generation methodology. These parameters are inspired from transML rule diagrams [45] to bridge the gap between DelTa models and transformations in the specific MTL: e.g., metamodel-specific type names. The editable parameter list is generated automatically from each DelTa model.

Following MDE practices, we have adopted the EMF [93] to create a domain-specific modeling environment for DelTa. With this modeling environment, we populated the catalog with design pattern structures conforming to the DelTa metamodel of Figure 3.1. This also gives the opportunity to define new design patterns or simply start designing a solution to a model transformation problem directly using DelTa. In order to show the applicability of our approach, we support two different MTLs: GrGen.NET and Henshin.

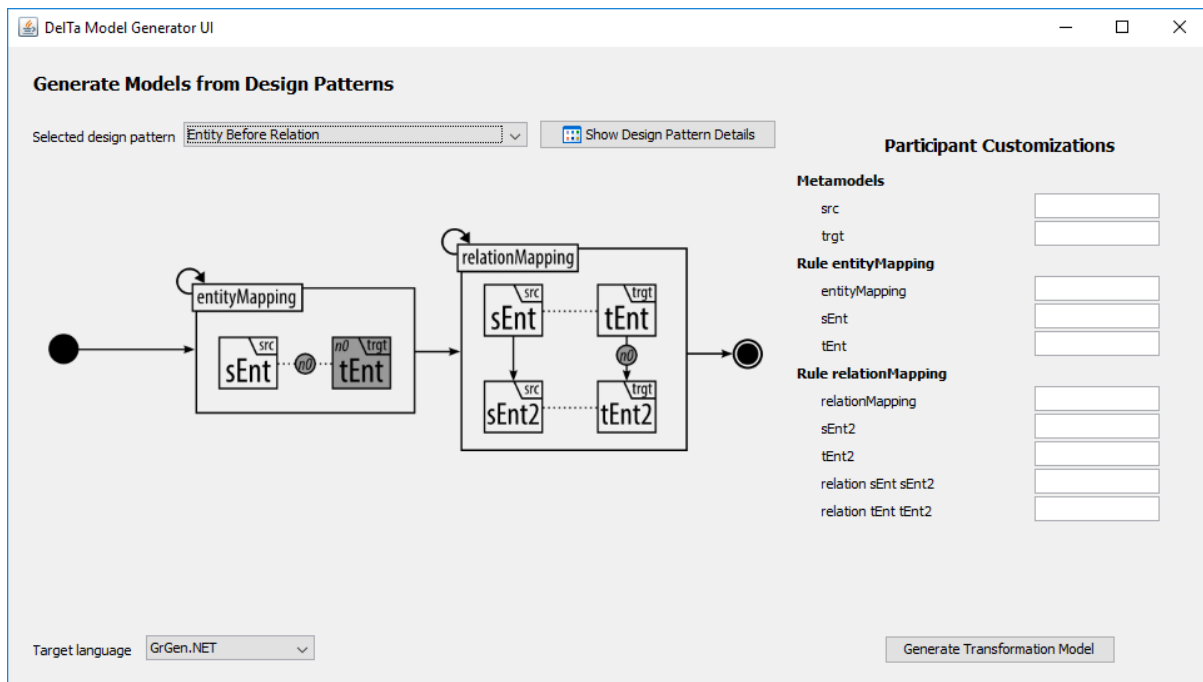


Figure 6.13: Design pattern generator tool

We choose these two languages because they are exogenous [23], which has different input and output metamodels. The choice of these two languages also demonstrates the breadth of applicability of DelTa across various MTLs.

6.5 Benefits of a Design Pattern-driven Methodology

The core object-oriented design patterns have already demonstrated multiple benefits. These include: (1) encapsulating the techniques to solve similar problems, (2) proposing a vocabulary that various domain experts can understand, and (3) improving the ability to document software by abstracting away the language details [1]. In the proposed design pattern driven development approach, we try to preserve these benefits as much as possible. We have observed that DelTa can assist in supporting the understanding of solution and documentation of modeling concerns. In the steps of our methodology, a model engineer can traverse existing patterns to find a solution to a specific problem by studying similar solutions. The automatic generation possibility directly from DelTa aids the model engineer by removing irrelevant implementation details and avoiding accidental complexities [16]. However, these benefits also result in several challenges. Studying the existing design patterns may also require additional effort, which adds additional time to solve a model transformation problem. After design patterns become more familiar by the domain experts, it is expected that this issue becomes less challenging.

6.6 Summary of the Methodology

In this chapter, we proposed a methodology that guides the developers on how to approach a model transformation problem using already existing design patterns. We illustrated the application of the methodology on a complex transformation from PNs to SCs. Then, we discussed alternative implementations for automatically generating concrete model transformations from design patterns. Finally, we described our tool to automate and guide model engineers throughout the process. The process and the tool required the user study in the next chapter to validate that they are useful for the community.

CHAPTER 7

VALIDATION: USER STUDY

In this chapter, we present the details of the user study we conducted to understand the validity of the design pattern driven development methodology of model transformations that we introduced in Chapter 6.

7.1 Objectives

In this study, we were specifically interested in the results of the following research questions:

RQ1 “Does the methodology have any impact on the model engineers who are trying to solve a model transformation problem?” From the survey we conducted in Section 3.1, it is apparent that model engineers use hand sketches or directly start implementing a model transformation solution in their favorite MTL. One possible reason is the lack of a methodology and we want to see the effect of the design pattern driven methodology on their way of solving a transformation problem.

RQ2 “Is the tool prototype given in Section 6.4 useful for model engineers?”

The same survey (i.e., in Section 3.1) also suggests that model engineers tend to use a designer tool if it exists when solving a problem using model transformation. The prototype lets them focus on solution candidates for identified subproblems and generate partial model transformations. This will help them spend less time working on the model transformation.

7.2 Experimental Setup

The study consisted of solving a model transformation problem using a specific MTL, GrGen.NET. We prepared a remote Windows machine hosted on Amazon EC2¹. The machine had all the necessary software installed and the participants only had to connect to the machine remotely and follow the directives. The directives page is designed as a web-based step-by-step tutorial that is also hosted on the same machine in order to direct the participants to the corresponding folders of each task they complete. Figure 7.1 depicts the welcome page of the directives.

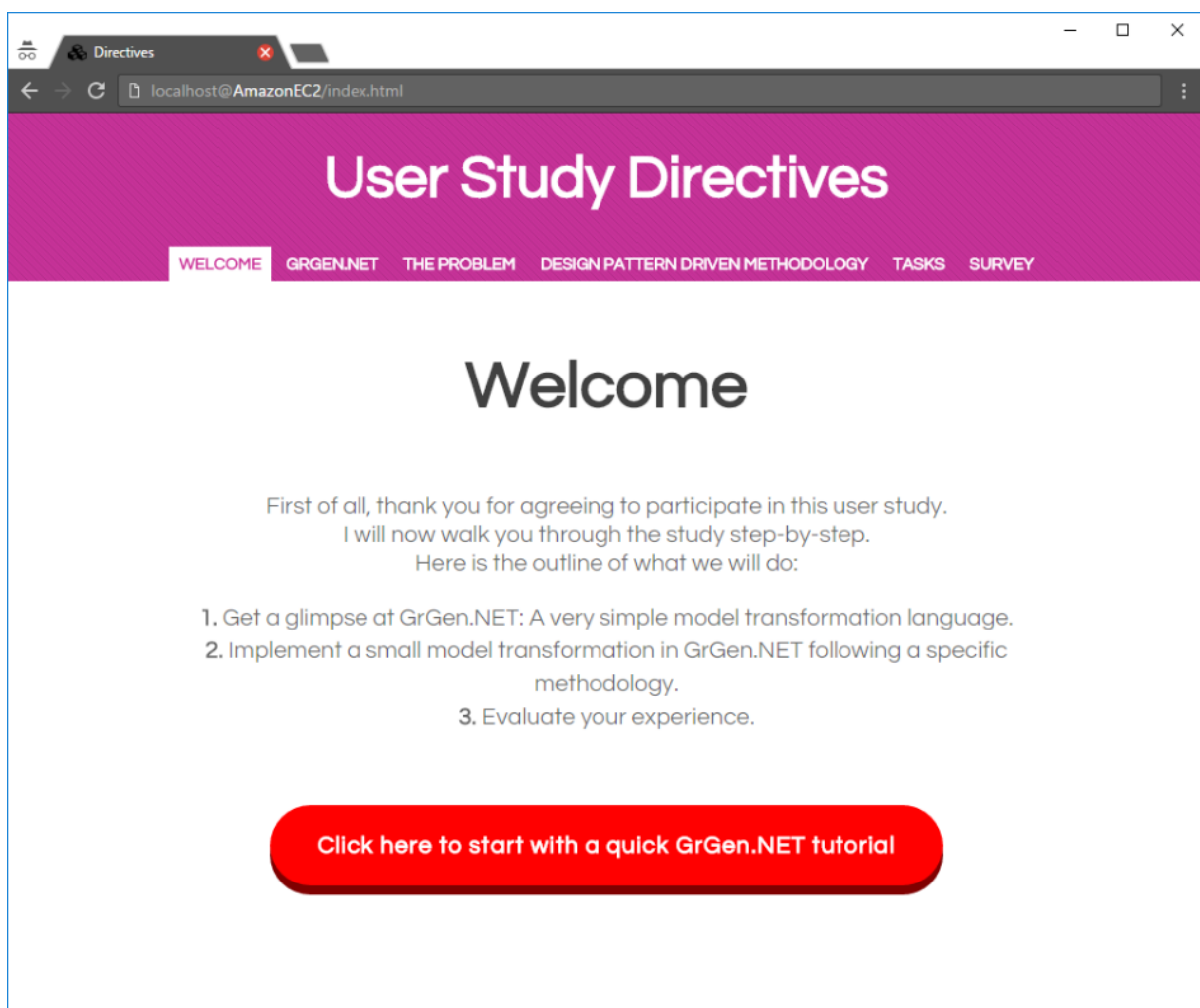


Figure 7.1: Welcome page

Participants had to walk through the following pages of the directives.

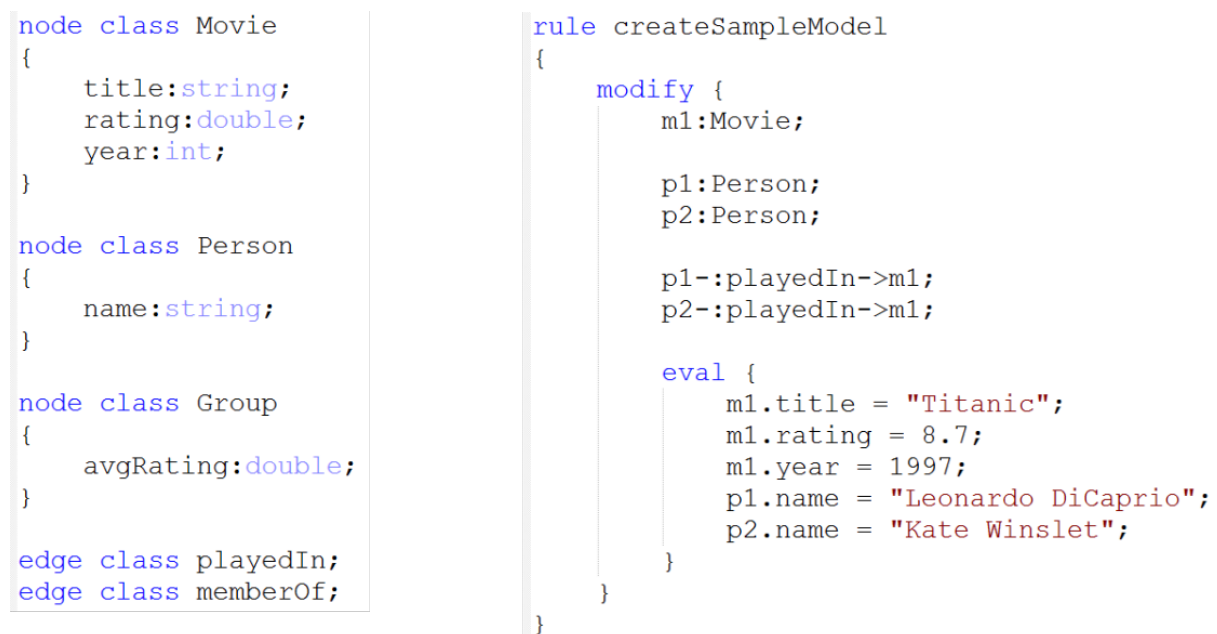
¹<http://aws.amazon.com/ec2/>

7.2.1 GrGen.NET Tutorial

This page covers the basics of the GrGen.NET model transformation language. The topics include metamodeling, transformation rules, scheduling, and creating models.

All participants were required to use a common MTL to ensure there is no bias. The simplicity of the language lowers the possible effect of not knowing what to implement. Therefore, we selected GrGen.NET for its reduced complexity and relatively faster learning curve among other MTLs.

Terminology is also supported with an example based on The Internet Movie Database (IMDB) case study [34]. In this case study, participants are provided with a metamodel that can represent actors, actresses, and movies, and a sample model created by a transformation rule. Figure 7.2 depicts the metamodel on the left and the rule that creates the sample model on the right.



```
node class Movie
{
    title:string;
    rating:double;
    year:int;
}

node class Person
{
    name:string;
}

node class Group
{
    avgRating:double;
}

edge class playedIn;
edge class memberOf;
```

```
rule createSampleModel
{
    modify {
        m1:Movie;

        p1:Person;
        p2:Person;

        p1-:playedIn->m1;
        p2-:playedIn->m1;

        eval {
            m1.title = "Titanic";
            m1.rating = 8.7;
            m1.year = 1997;
            p1.name = "Leonardo DiCaprio";
            p2.name = "Kate Winslet";
        }
    }
}
```

Figure 7.2: IMDB Metamodel and Sample Model Creation Rule

Figure 7.3 depicts the rule we have executed on the sample model to create a group for the “people who played on the same movie together” and assign an average rating to the group. The top-right of the figure is the visual representation of the sample model before the transformation is executed and the bottom-right is the output of the transformation that has a new element “group” for the two people played on the same movie “Titanic.”

```
rule createGroups
```

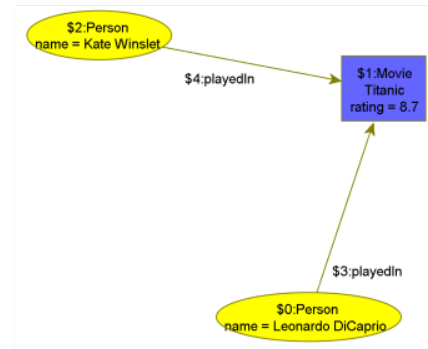
```
{
  m1:Movie;
  p1:Person;
  p2:Person;
  p1-:playedIn->m1;
  p2-:playedIn->m1;

```

```
negative {
  g1:Group;
  p1-:memberOf->g1;
  p2-:memberOf->g1;
}
```

```
modify {
  g1:Group;
  p1-:memberOf->g1;
  p2-:memberOf->g1;
  eval {
    g1.avgRating = m1.rating;
  }
}
```

Sample Model



After Transformation

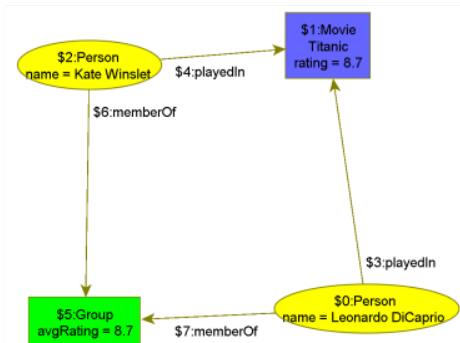


Figure 7.3: IMDB CreateGroups Rule and The Sample Model Before and After the Transformation

7.2.2 The Problem

This page describes the problem in detail. We provided the participants the simplified metamodels of C and Java depicted in Figure 7.4 The core problem is to transform the *struct* and *pointers* in C to a hierarchy of Java *classes*. The problem is easy to understand,

```
// A very simple C Struct

node class Struct
{
  name:string;
}
edge class pointer;
```

```
// A very simple Java Class
```

```
node class Class
{
  name:string;
  depth:int = 0;
  process:boolean = false;
  mark:boolean = false;
}

edge class inheritance;
edge class association;
edge class composition;
```

Figure 7.4: Simplified C and Java Metamodels

but not trivial to implement because, in the transformation process, one should consider that Java does not support multiple inheritance. Therefore, only one pointer link between two structs should be transformed into inheritance relations in Java. All additional links should be transformed into associations. To increase the complexity of the problem, the transformation should also compute the depth level of each class in the output class hierarchy as a second task. The depth level represents the number of classes between a class and its furthest ancestor.

Figure 7.5 depicts an input to the problem on the top, which is a group of C structs connected with pointers, and the output of the same input on the bottom, which is Java class equivalents of the C structs and inheritances/associations instead of pointers. The green links are traceability links to connect the source elements to the target element.

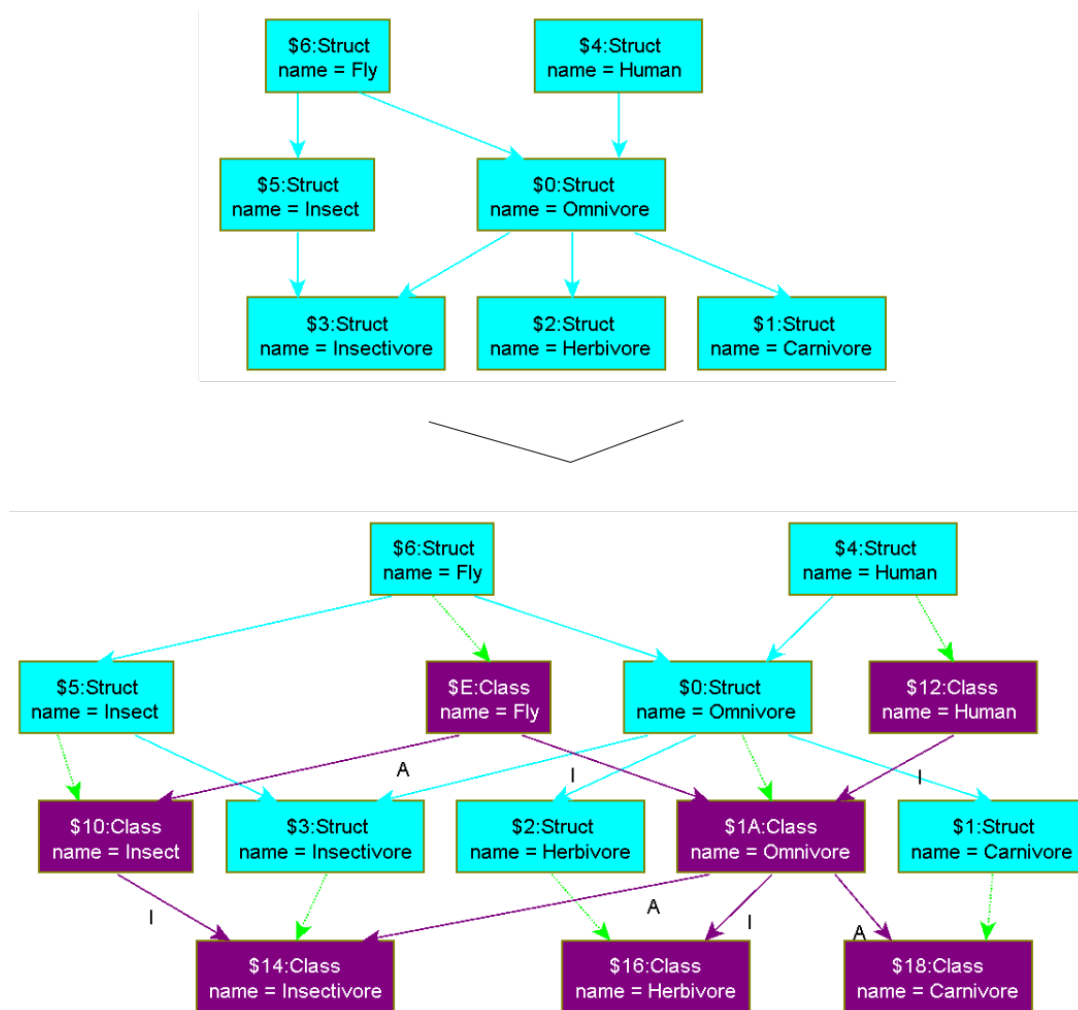


Figure 7.5: The Sample Input and the Output

7.2.3 The Methodology

This page summarizes the 5 steps of the methodology introduced in Chapter 6. Then, it presents a small tutorial on DelTa based on Figure 3.2 and associated text in Section 3.3.2. Finally, it presents a tutorial on how to use the tool introduced in Section 6.4. We have used the same example of IMDB case study to train participants about the tool [34]. We have chosen not to depend on a design pattern when introducing DelTa and relied on a different problem example in order to avoid any bias when completing the tasks. Therefore, we designed a new dummy design pattern that our tool can generate the *createGroups* rule in Figure 7.3. The dummy design pattern and necessary customizations to generate the rule are presented in Figure 7.6

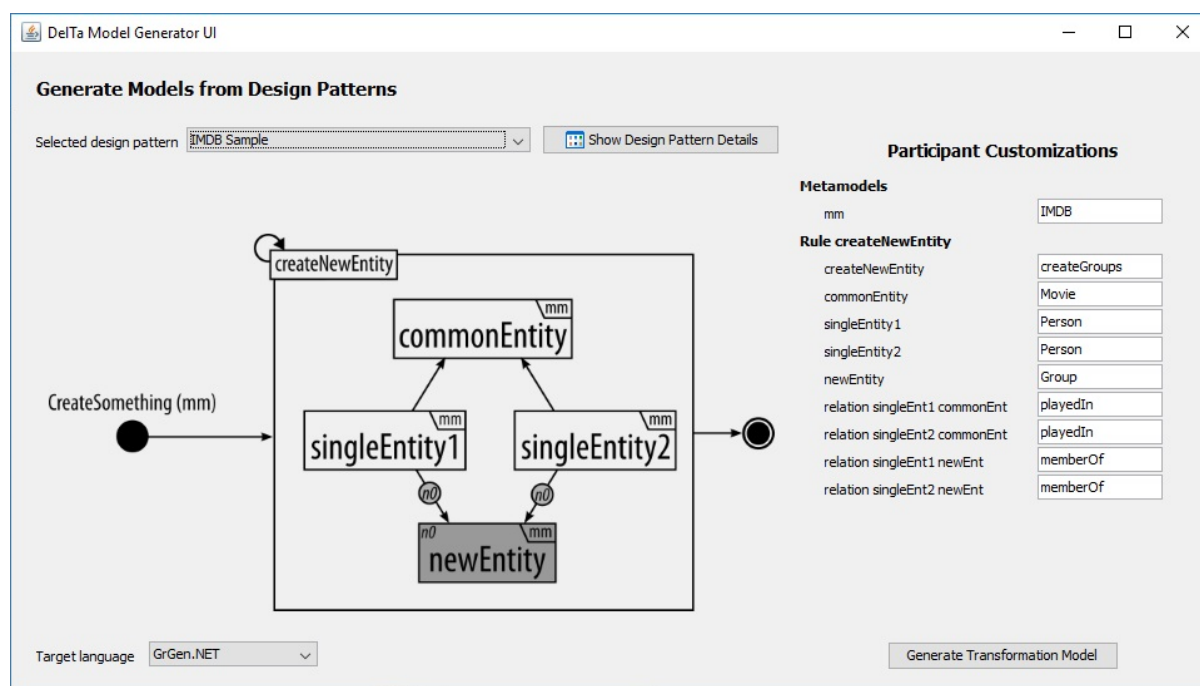


Figure 7.6: The Dummy Pattern to Use in Training of the Tool

7.2.4 Tasks

This page lists the tasks the participants have to complete. We asked them to complete two tasks:

- The first task is about mapping. C structs have access to all variables in other structs as long as they have pointers to the other structs. In the sample problem, we interpret pointers as inheritance links between classes. However, Java does not

allow for multiple inheritance. Therefore, the participants need to make sure that additional inheritance links are mapped to other types of relations between classes.

- The second task is to compute the depth level of each class in the Java class inheritance hierarchy. The depth level of a base class (not inheriting from any other class) is 0. For example, if C inherits from B and B inherits from A, then C will have a depth level 2, B will have 1 and A will have 0.

We have set up in advance all necessary folders and batch scripts to run and test the transformations easily. Therefore, the users can focus on the task of working through the solution and evaluating the methodology, the tool and DelTa.

7.2.5 Survey

In this page, participants answer the questions of a survey related to their experience. The survey consists of 5 questions and a free form text for feedback comments. We ask a question about their experience with GrGen.NET. Then, we ask them whether the methodology had any impact on their conceptual thinking for a solution to the problem. When they go over the patterns provided in the tool, it should be easier for them to create a solution to the problem. The remaining questions ask the participants to rate various properties of the methodology, the tool, and the DelTa language on a 5-point scale (i.e., “bad,” “poor,” “average,” “good,” and “excellent”). The properties include understandability, readability, usefulness, appropriateness, completeness, and fitness.

Participants had a two-hour time slot allotted to them with access to resources we provided at any time. We provided participants with initial projects, along with the tools necessary for training purposes.

7.3 Data Collection

We collected and analyzed the actual transformation solutions after each participant completed his/her study. The post-survey is in Appendix C. We used the Qualtrics software to collect and analyze the results of the survey. The complete results of the survey are presented in Appendix C.

7.4 Participant Selection

We selected participants from people who have developed model transformation in the past. Among the participants of the first pilot survey, two joined this study. In total, 10 academic people participated in this study. Only one of them declared he had used GrGen.NET before, which gave us the opportunity to analyze the effects of the methodology on participants who never used this MTL before.

7.5 Results of the User Study

Had impact	Result
Yes	7
No	3

Table 7.1: Effect of the methodology

Task	Result
First (Translation)	90%
Second (Depth Level)	30%

Table 7.2: Task completion ratio

Question	Rating	Rated 4-5
About methodology		
Did you understand it?	4.4	90%
Is it useful?	4.1	80%
Do you find it natural?	3.4	50%
Would you adopt it in the future?	3.7	70%
About DelTa		
Understandability of design patterns	3.7	60%
Readability of design patterns	4.4	80%
Usefulness	4	70%
Appropriateness	4.4	90%
Completeness	3.6	60%
About the tool		
Easiness to use	4.2	80%
Intuitiveness	4.1	80%
Usefulness	4.3	80%
Correctness	4.5	100%

Table 7.3: Ratings of the properties

RQ1: Impact of the methodology on model engineers The setup was such that participants were first given the problem, and only then was the methodology and design patterns revealed with minimal training. When solving the problem, participants had to choose the most appropriate design patterns from the ones available in the tool.

This setup was to reduce the probability of bias with the methodology when asked about it.

Table 7.3 summarizes how participants rated various properties of the methodology, the DelTa language, and the tool. We show the average ratings of each question in the second column. We also show what percentage of the participants rated a property with “excellent” or “good” in the third column. Although most participants understood the methodology and found it useful, half of the participants found the methodology natural. The same participants who felt the methodology impacted their conceptual solution said they would reuse it in the future.

Table 7.1 shows that 7 out of 10 participants acknowledged that the methodology had a positive impact on how they approached the solution to the problem. The methodology helped them implement a transformation from scratch successfully in a language that was completely new to them. The remaining three stated that they did not need the methodology to be able to solve the problem. Nevertheless, after examining their transformation, the solution was no different from those who claimed it did. In fact, they followed the methodology even though they claimed it did not influence them.

Table 7.2 reports how many participants completed each task successfully. 90% of the participants were able to solve the first task using the automatic generation capability of the tool, after examining the problem and the seeking for the required pattern to be used. However, only 30% were able to complete the second task. Although this task was a bit harder, all participants stated that the limited time prevented them from completing it.

RQ2: Usefulness of the design pattern generator tool The tool generates a partial transformation from a selected design pattern. In addition to the usefulness of code generation, such as focusing on the overall structure instead of implementation details, the tool also provides a comprehensive catalog to explore design patterns. Participants had to choose the right design pattern, generate the partial GrGen.NET code and manually refine the transformation to solve the problem correctly.

Table 7.3 shows that most participants found the tool to be very useful, easy, and intuitive to use. Furthermore, all participants agreed that the generated transformation is

correct, which validates our own test results. DelTa, as the language of pattern structure, was also well-appreciated in terms of readability and usefulness. Also, all participants agreed that DelTa offers an appropriate representation and description of the structure of design patterns. This concurs with the results of the former pilot survey in Section 3.1. 40% of the participants did not understand very well the patterns because they did not click on the description button to read the complete specification of the design patterns. The DelTa model is only one part of the design pattern definition, but participants relied only on the DelTa model of the pattern rather than checking the whole characteristics. The same participants also questioned the ability of DelTa to cover all possible design patterns (completeness). One possible explanation is that the prototype they were given only listed five design patterns from the catalog. We made this choice to reduce the amount of reading for participants due to the time limit.

7.6 Threats to Validity

There are various threats to the validity of this empirical study. Threats to internal validity include the longer training session at the beginning of the study. We tried to eliminate this threat by making the training as simple as possible in the directives file. However, this was a trade-off to impose a time limit of two hours. We feel that allowing more time to solve the problem may have exhausted some participants who would have then dropped out of the user study.

The same threats to external validity of the motivational survey in Section 3.1 applies here as all our participants are from an academic background. Another threat is about the number of participants and how far we can generalize the results. In addition, we assumed all participants are familiar with object-oriented design patterns and can easily continue with model transformation design patterns. Some participants ended up not knowing about the object-oriented design patterns. However, they still solved the tasks. We should also note that this was the most of the participants' first exposure to the DelTa and model transformation design patterns.

Finally, some participants did not follow the tutorials. Therefore, they chose a harder

way to understand each design pattern, which is by structure only, instead of a full description.

7.7 Summary of the Validation

In this chapter, we describe the results of a user study to validate the methodology combined with the tool and the language, DelTa, for model transformation design patterns. The overall results of the survey, in which the tool, the language, and the methodology are rated, are promising. When we check the concrete transformations that the participants implemented, we see that they all used the tool in order to solve the problem. In summary, even though the participants had to put much effort into completing the study, they agreed that the methodology is useful when implementing model transformations.

CHAPTER 8

CONCLUSION AND FUTURE WORK

We conclude by summarizing the contributions of this dissertation and outlining future work. The work presented in this dissertation makes several contributions to the field of MDE and, in particular, model transformations.

8.1 Summary and Contributions

After analyzing existing model transformation design pattern studies in Chapter 2, we discovered that some should not have been classified as design patterns. We also noted no consensus on how to represent model transformation design patterns. In Chapter 3, we surveyed model transformation engineers to understand the needs for model transformation design patterns and the essential requirements for a language to express them. From the results of this survey, we created a unified template to express model transformation design patterns and a language to support the solution of the pattern. DelTa fulfills the initial requirements in that it is a language for describing patterns rather than transformations, it is independent from any MTL yet directly implementable in most MTLs. We showed that the unified template supported with DelTa can be used to express all 13 existing design patterns in the catalog presented in Chapter 5. We also introduced two new design patterns: *Fixed-point Iteration* (Chapter 4) and *Execution by Translation* (Chapter 5), as well as generalizing 13 existing ones. A follow-up informal survey we conducted with the same participants showed preliminary validation that DelTa is an appropriate DSL to express model transformation design patterns, it is easily understandable by model engineers, and can be used directly in their implementation processes. In Chapter 6, we proposed a methodology to produce model transformations focusing on design patterns by selecting, customizing, and generating partial implementations. We

also implemented a tool that assists model engineers in following this methodology to help guide model engineers in their design and implementation. Finally, we evaluated the methodology, the tool, and DelTa with a user study that showed clear improvements in the design of model transformations in Chapter 7.

8.2 Future Uses of DelTa

We foresee several uses of DelTa in the future. First, DelTa can be used to document design patterns. Model engineers can refer to the catalog in Chapter 5 to learn and understand model transformation design patterns. As witnessed in both user studies, the syntax of DelTa is intuitive to model engineers. Therefore, we are confident that DelTa will facilitate the comprehension and adoption of design patterns in future model transformation implementations.

Second, we showed in Chapter 6 that design patterns defined in DelTa are directly implementable. Model transformations can be automatically generated from DelTa models. Similar to how UML is often used by software engineers to design and implement object-oriented programs, we foresee DelTa being used by model engineers to design and implement model transformations following the methodology we propose. The architecture of the prototype we developed facilitates the generation of model transformations in a variety of MTLs.

Third, DelTa can be used to verify whether a given model transformation correctly implements a design pattern. Detecting correct or ill-formed instances of design patterns is very helpful to increase the quality of existing implementations [103]. One possibility to achieve this with model transformations is to translate a concrete model transformation implementation into a DelTa model that abstracts its essence. This model can be compared with individual design patterns in DelTa by filtering elements that are not required in the design pattern and output an approximate correspondence between the abstract DelTa model of the transformation and the design pattern.

8.3 Future Work

Our implementation was demonstrated to work well with model transformation languages based on graph transformation. It would be interesting to investigate how automatic generation of instances of design patterns can be extended to other model transformation approaches: exogenous model-to-model transformations, such as QVT-Operational Mappings [65] and ATL [54], and bi-directional transformations, such as QVT-R [65] and Triple Graph Grammars [43].

Furthermore, most design patterns presented in the catalog are only applicable to in-place transformations. However, since the majority of problems solved by model transformations are exogenous [30], we need to further investigate design patterns applicable to these kinds of problems. Although the initial study in Chapter 7 shows promising results, a more extensive community-wide study is necessary to further understand the benefits and disadvantages of the design pattern driven methodology. However, as we discovered in the feedback of the study, it is important that participants of the study should already be trained with design patterns for model transformations in order to focus on the methodology and eliminate the most of the background training. Therefore, it would be ideal to integrate the findings of this user study in advanced MDE courses.

Finally, as pointed out in Section 8.2, the verification of correct implementations of a design pattern in a concrete model transformation still remains. This would have tremendous benefits to model engineers by providing them with feedback on the quality of their transformation through corrective suggestions.

The detection of design pattern instances in model transformations are also an interesting area that can be further studied. DelTa is a fully modeled language and detection requires another HOT in order to find the instances of DelTa models in concrete model transformations. Mokaddem et al. [81] initiated a study for this purpose and the results are promising. Dong et al. [53] studied a comprehensive review on other detection techniques, which can also be adopted by model transformations.

REFERENCES

- [1] Ellen Agerbo and Aino Cornils. How to Preserve the Benefits of Design Patterns. *ACM SIGPLAN Notices*, 33(10):134–143, October 1998.
- [2] Aditya Agrawal. Reusable Idioms and Patterns in Graph Transformation Languages. In *International Workshop on Graph-Based Tools*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 181–192, 2005.
- [3] Aditya Agrawal, Gabor Karsai, and Feng Shi. Graph Transformations on Domain-specific Models. *Journal on Software and Systems Modeling*, 37:1–43, 2003.
- [4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. On Finding Lowest Common Ancestors in Trees. *SIAM Journal on Computing*, 5(1):115–132, 1976.
- [5] Herv Albin Amiot, Pierre Cointe, Yann-Gal Guhneuc, and Narendra Jussien. Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In *Automated Software Engineering*, pages 166–173, Coronado Island, San Diego, CA, 2001.
- [6] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.
- [7] Sergio Antoy and Michael Hanus. New Functional Logic Design Patterns. In *20th International Workshop on Functional and Constraint Logic Programming*, pages 19–34, Odense, Denmark, July 2011.
- [8] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 121–135. Oslo, Norway, 2010.
- [9] Yariv Aridor and Danny B. Lange. Agent Design Patterns: Elements of Agent Application Design. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 108–115, Minneapolis, Minnesota, 1998.
- [10] Anya Helene Bagge and Ralf Lämmel. Walk Your Tree Any Way You Want. In *6th International Conference on Theory and Practice of Model Transformations*, pages 33–49, Budapest, Hungary, June 2013.

- [11] Paul Baker, Shiou Loh, and Frank Weil. Model-Driven Engineering in a Large Industrial Context - Motorola Case Study. In *8th International Conference Model Driven Engineering Languages and Systems*, pages 476–491, Montego Bay, Jamaica, October 2005.
- [12] Kent Beck and Ralph E. Johnson. Patterns Generate Architectures. In *Object-Oriented Programming, Proceedings of the 8th European Conference*, pages 139–149, Bologna, Italy, July 1994.
- [13] Jean Bézivin, Frédéric Jouault, and Jean Paliès. Towards Model Transformation Design Patterns. In *Proceedings of the First European Workshop on Model Transformations*, page 116, 2005.
- [14] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
- [15] Ghizlane El Boussaidi and Hafedh Mili. A Model-driven Framework for Representing and Applying Design Patterns. In *31st Annual International Computer Software and Applications Conference*, volume 1, pages 97–100, Beijing, China, July 2007.
- [16] Frederick P. Brooks, Jr. No Silver Bullet Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.
- [17] William Brown, Raphael Malveau, Skip McCormick, and Tom Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.
- [18] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. Automatic Code Generation from Design Patterns. *IBM Systems Journal*, 35(2):151–171, 1996.
- [19] Frank Buschmann, Kevin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Wiley, 2007.
- [20] Frank Buschmann, Reginem Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern Oriented Software Architecture. A System of Patterns*, volume 1. 1996.
- [21] Hyun Cho and Jeff Gray. Design Patterns for Metamodels. In *SPLASH '11 Domain-specific Modeling Workshop*, pages 25–32, Portland, OR, 2011.

- [22] Alexandre L. Correa, Cláudia M. L. Werner, and Gerson Zaverucha. Object Oriented Design Expertise Reuse: An Approach Based on Heuristics, Design Patterns and Anti-patterns. In *Software Reuse: Advances in Software Reusability*, volume 1844 of *LNCs*, pages 336–352, Vienna, Austria, June 2000.
- [23] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, July 2006.
- [24] Artur Czumaj, Mirosław Kowaluk, and Andrzej Lingas. Faster Algorithms for Finding Lowest Common Ancestors in Directed Acyclic Graphs. *Theoretical Computer Science*, 380(1):37–46, 2007.
- [25] Juan de Lara and Hans Vangheluwe. Automating the Transformation-based Analysis of Visual Languages. *Formal Aspects of Computing*, 22(3-4):297–326, 2010.
- [26] Andre DeHon, Jacob Adams, Michael deLorimier, Nachiket Kapre, Yoshio Matsuda, Helia Naeimi, Michael Vanier, and Michael Wrighton. Design Patterns for Reconfigurable Computing. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 13–23, Napa, CA, April 2004.
- [27] Edsger W Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [28] Bruce Powell Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [29] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the International Conference on Software Engineering*, pages 411–420, Los Angeles, CA, May 1999.
- [30] Edouard Batot, Houari Sahraoui, Eugene Syriani, Paul Molins, and Wael Sboui. Systematic Mapping Study of Model Transformations for Concrete Problems. In *Model-Driven Engineering and Software Development*, pages 176–183. SciTePress, 2016.
- [31] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006.
- [32] Huseyin Ergin and Eugene Syriani. AToMPM Solution for the Petri Net to Statecharts Case Study. In *Seventh Transformation Tool Contest*, Budapest, Hungary, July 2013.

- [33] Huseyin Ergin and Eugene Syriani. Identification and Application of a Model Transformation Design Pattern. In *ACM Southeast Conference*, Savannah, GA, April 2013.
- [34] Huseyin Ergin and Eugene Syriani. AToMPM Solution for the IMDB Case Study. In *Seventh Transformation Tool Contest*, pages 134–138, York, UK, 2014.
- [35] Huseyin Ergin and Eugene Syriani. Implementations of Model Transformation Design Patterns Expressed in DelTa. Tech report SERG-2014-01, University of Alabama, Department of Computer Science, February 2014.
- [36] Huseyin Ergin and Eugene Syriani. Towards a Language for Graph-Based Model Transformation Design Patterns. In *Theory and Practice of Model Transformations*, volume 8568 of *LNCS*, pages 91–105, York, UK, 2014.
- [37] Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 307–309, Reno/Tahoe, Nevada, October 2010.
- [38] Jean-Rémy Falleri, Marianne Huchard, and Clementine Nebut. Towards a Traceability Framework for Model Transformations in Kermeta. In *European Conference on Model-Driven Architecture Traceability Workshop*, pages 31–40, Bilbao, Spain, 2006.
- [39] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In *10th International Conference Model Driven Engineering Languages and Systems*, pages 482–497, Nashville, TN, October 2007.
- [40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.
- [41] Aldo Gangemi and Valentina Presutti. *Ontology Design Patterns*, pages 221–243. 2009.
- [42] Rubino Geiß and Moritz Kroll. GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *LNCS*, pages 568–569. Kassel, Germany, October 2008.

- [43] Joel Greenyer and Ekkart Kindler. Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. *Software & Systems Modeling*, 9(1):21–46, January 2010.
- [44] Object Management Group. *Meta Object Facility (MOF) Core Specification*, 2006.
- [45] Esther Guerra, Juan de Lara, Dimitrios Kolovos, Richard Paige, and Osmar dos Santos. Engineering Model Transformations with transML. *Journal on Software and Systems Modeling*, 12(3):555–577, July 2011.
- [46] David Harel and Bernhard Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Technical report, Weizmann Institute Of Science, 2000.
- [47] Dov Harel and Robert Endre Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984.
- [48] Seyed Mohammad Hossein Hasheminejad and Saeed Jalili. Design Patterns Selection: An Automatic Two-phase Method. *Journal on Software and Systems Modeling*, 85(2):408–424, February 2012.
- [49] Brian Huston. The Effects of Design Pattern Application on Metric Scores. *Journal of Systems and Software*, 58(3):261–269, 2001.
- [50] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical Assessment of MDE in Industry. In *33rd International Conference on Software Engineering*, pages 471–480, Honolulu, HI, May 2011.
- [51] John Edward Hutchinson, Jon Whittle, and Mark Rouncefield. Model-Driven Engineering Practices in Industry: Social, Organizational and Managerial Factors That Lead to Success or Failure. *Science of Computer Programming*, 89:144–161, 2014.
- [52] Maria-Eugenia Iacob, Maarten W. A. Steen, and Lex Heerink. Reusable Model Transformation Patterns. In *Enterprise Distributed Object Computing Workshop*, pages 1–10, Munich, Germany, September 2008.
- [53] Jing Dong, Yajing Zhao, and Tu Peng. A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering*, 19(6):823–855, 2009.
- [54] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72(1-2):31–39, 2008.

- [55] Frédéric Jouault and Ivan Kurtev. On the Interoperability of Model-to-Model Transformation Languages. *Science of Computer Programming, Special Issue on Model Transformation*, 68(3):114–137, October 2007.
- [56] Stuart Kent. Model Driven Engineering. *Integrated Formal Methods*, 2335(2):286–298, 2002.
- [57] Foutse Khomh and Yann-Gael Gueheneuc. Do Design Patterns Impact Software Quality Positively? In *12th European Conference on Software Maintenance and Reengineering*, pages 274–278, Athens, Greece, April 2008.
- [58] Dae-Kyoo Kim, Lunjin Lu, and Byunghun Lee. Design Pattern-based Model Transformation Supported by QVT. *Journal of Systems and Software*, 125:289–308, 2017.
- [59] Benjamin Klatt. Xpand: A Closer Look at the Model2text Transformation Language. *Language*, 10(16), 2007.
- [60] Thomas Klein, Ulrich Nickel, Jörg Niere, and Albert Zündorf. From UML to Java and Back Again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, September 1999.
- [61] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture: Practice And Promise*. Addison-Wesley, 2003.
- [62] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Transformation Language. In *International Conference on Model Transformations*, volume 5063 of *LNCS*, pages 46–60. Zürich, Switzerland, 2008.
- [63] Glenn E. Krasner and Stephen T. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.
- [64] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit Transformation Modeling. In *Models in Software Engineering Workshop*, volume 6002 of *LNCS*, pages 240–255, Denver, CO, 2010.
- [65] Ivan Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *LNCS*, pages 377–393. Kassel, Germany, 2008.

- [66] Ralf Lämmel and Simon L. Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *Proceedings of ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 26–37, New Orleans, LA, January 2003.
- [67] Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic Programming Meets Adaptive Programming. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 168–177, Boston, MA, March 2003.
- [68] Ralf Lämmel and Joost Visser. Design patterns for functional strategic programming. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming*, pages 1–14, 2002.
- [69] Kevin Lano, David Clark, and Kelly Androutsopoulos. RSDS, a Subset of UML with Precise Semantics. *L’OBJET*, 9(4):53–73, 2003.
- [70] Kevin Lano and Shekoufeh Kolahdouz Rahimi. Constraint-based Specification of Model Transformations. *Journal of Systems and Software*, 86(2):412–436, 2013.
- [71] Kevin Lano and Shekoufeh Kolahdouz Rahimi. Model-Transformation Design Patterns. *IEEE Transactions on Software Engineering*, 40(12):1224–1259, 2014.
- [72] Alain Le Guennec, Gerson Sunye, and Jean-Marc Jezequel. Precise Modeling of Design Patterns. In *The Unified Modeling Language*, volume 1939 of *LNCIS*, pages 482–496. York, UK, 2000.
- [73] Laszló Lengyel, Tihamér Levendovszky, Gergely Mezei, and Hassan Charaf. Model Transformation with a Visual Control Flow Language. *International Journal of Computer Science*, 1(1):45–53, 2006.
- [74] Tihamér Levendovszky, Laszló Lengyel, and Tamós Mészáros. Supporting Domain-specific Model Patterns with Metamodeling. *Journal on Software and Systems Modeling*, 8(4):501–520, 2009.
- [75] Levi Lúcio, Moussa Amrani, Juergen Dingel, Leen Lambers, Rick Salay, Gehan M. K. Selim, Eugene Syriani, and Manuel Wimmer. Model Transformation Intents and Their Properties. *Journal on Software and Systems Modeling*, 15(3):647–684, 2016.
- [76] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.

- [77] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Software Patterns Series. Pearson Education, 2004.
- [78] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.
- [79] Parastoo Mohagheghi and Vegard Dehlen. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In *4th European Conference Model Driven Architecture - Foundations and Applications*, pages 432–443, Berlin, Germany, June 2008.
- [80] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A. Fernandez. An Empirical Study of the State of the Practice and Acceptance of Model-Driven Engineering in Four Industrial Cases. *Empirical Software Engineering*, 18(1):89–116, 2013.
- [81] Chihab Eddine Mokaddem, Houari Sahraoui, and Eugene Syriani. Towards Rule-Based Detection of Design Patterns in Model Transformations. In *9th International Conference System Analysis and Modeling. Technology-Specific Aspects of Models*, pages 211–225, Saint-Melo, France, October 2016.
- [82] Thomas J. Mowbray and Raphael C. Malveau. *CORBA Design Patterns*. Wiley, 1997.
- [83] Object Management Group. *Meta Object Facility 2.0 Query/View/Transformation Specification*, Jan 2011.
- [84] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F. Tichy. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Transactions on Software Engineering*, 28(6):595–606, June 2002.
- [85] Frank Radeke, Peter Forbrig, Ahmed Seffah, and Daniel Sinnig. Pim Tool: Support for Pattern-Driven and Model-Based UI Development. In *5th International Workshop Task Models and Diagrams for Users Interface Design*, pages 82–96, Hasselt, Belgium, October 2006.
- [86] Arend Rensink and Pieter Van Gorp. Graph Transformation Tool Contest 2008. *International Journal on Software Tools for Technology Transfer*, 12(3-4):171–181, 2010.

- [87] Baruch Schieber and Uzi Vishkin. On Finding Lowest Common Ancestors: Simplification and Parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.
- [88] Benedikt Schulz, Thomas Genssler, Berthold Mohr, and Walter Zimmer. On the Computer Aided Introduction of Design Patterns into Object-oriented Systems. In *Proceedings Technology of Object-Oriented Languages*, pages 258–267, September 1998.
- [89] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.
- [90] Diomidis Spinellis. Notable Design Patterns for Domain-specific Languages. *Journal of Systems and Software*, 56(1):91–99, 2001.
- [91] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
- [92] Mirosław Staron. Adopting Model Driven Software Development in Industry - A Case Study at Two Companies. In *9th International Conference on Model Driven Engineering Languages and Systems*, pages 57–72, Genova, Italy, October 2006.
- [93] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.
- [94] Eugene Syriani and Huseyin Ergin. Operational Semantics of UML Activity Diagram: An Application in Project Management. In *Requirements Engineering Conference Workshops: Model-driven Requirements Engineering*, pages 1–8, Chicago, IL, 2012.
- [95] Eugene Syriani and Jeff Gray. Challenges for Addressing Quality Factors in Model Transformation. In *Software Testing, Verification and Validation*, pages 929–937, Montreal, QC, April 2012.
- [96] Eugene Syriani, Jeff Gray, and Hans Vangheluwe. Modeling a Model Transformation Language. In *Domain Engineering: Product Lines, Conceptual Models, and Languages*, pages 211–237. 2012.
- [97] Eugene Syriani and Hans Vangheluwe. De-/Re-constructing Model Transformation Languages. *European Association of Software Science and Technology*, 29, March 2010.

- [98] Eugene Syriani and Hans Vangheluwe. A Modular Timed Graph Transformation Language for Simulation-based Design. *Journal on Software and Systems Modeling*, 12(2):387–414, 2013.
- [99] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. AToMPM: A Web-based Modeling Environment. In *Joint Proceedings of MODELS’13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems*, pages 21–25, September 2013.
- [100] Gabriele Taentzer. AGG: A tool environment for algebraic graph transformation. In *Applications of Graph Transformations with Industrial Relevance, International Workshop*, pages 481–488, Kerkrade, The Netherlands, September 1999.
- [101] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *European Conference on Model Driven Architecture: Foundations and Applications*, volume 5562 of *LNCS*, pages 18–33, Enschede, The Netherlands, June 2009.
- [102] Lance Tokuda and Batory Don. Evolving Object-Oriented Designs with Refactorings. *Automated Software Engineering*, 8(1):89–120, 2001.
- [103] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, November 2006.
- [104] Pieter Van Gorp and Louis M. Rose. The Petri-Nets to Statecharts Transformation Case. In *Sixth Transformation Tool Contest*, volume 135, pages 16–31, Budapest, Hungary, June 2013.
- [105] Daniel Varro. Model Transformation by Example. In *Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 410–424. Genova, Italy, 2006.
- [106] Daniel Varro and Andras Balogh. The Model Transformation Language of the VIATRA2 Framework. *Science of Computer Programming*, 68(3):214–234, 2007.
- [107] Eelco Visser. A Survey of Rewriting Strategies in Program Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 57:109–143, 2001.
- [108] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In *Rewriting Techniques and*

Applications, volume 2051 of *LNCS*, pages 357–361. Utrecht, The Netherlands, May 2001.

- [109] Sally K. Wahba, Jason O. Hallstrom, and Neelam Soundarajan. Initiating a Design Pattern Catalog for Embedded Network Systems. In *Proceedings of the 10th International Conference on Embedded Software*, pages 249–258, Scottsdale, Arizona, October 2010.
- [110] Xue B. Wang, Quan Y. Wu, Huai M. Wang, and Dian X. Shi. Research and Implementation of Design Pattern-oriented Model Transformation. In *International Multi-Conference on Computing in the Global Information Technology*, pages 24–24, March 2007.
- [111] Sherif M. Yacoub and Hany Hussein Ammar. *Pattern-oriented Analysis and Design: Composing Patterns to Design Software Systems*. Addison-Wesley Professional, 2004.
- [112] Bahman Zamani, Greg Butler, and Sahar Kayhani. Tool Support for Pattern Selection and Use. *Electronic Notes in Theoretical Computer Science*, 233:127–142, March 2009.

APPENDIX A

MOTIVATIONAL SURVEY QUESTIONS & RESULTS

This appendix has all the questions and results of the motivational survey to understand whether the community needs a design pattern language in Section 3.1. We used Qualtrics to distribute the survey and collect the results. In the free-form questions, all grammatical errors of the participants are left as is for authenticity.

A.1 Background Information

We asked the following questions to the participants in order to understand their backgrounds.

A.1.1 Have you ever written a model transformation?

Answers	Count	Percentage
Yes	22	96%
No	1	4%

Table A.1: Results of Question A.1.1

A.1.2 What is the primary focus of your software development?

Answers	Count	Percentage
Academic Work	20	95%
Industrial Work	1	5%
Hobby	0	0%

Table A.2: Results of Question A.1.2

A.1.3 How much time on average do you spend on the design phase of a typical software project, as opposed to planning, implementation, testing etc.?

Answers	Min	Max	Average	Responses
Percentage	7%	80%	27.59%	22

Table A.3: Results of Question A.1.3

A.1.4 Do you use the following tools for designing and in what ratio? Please complete to a 100% in total.

Answers	Average
UML Tools (e.g. Enterprise Architect, Rational Rose, Astah UML, Visio, UMLet or others.)	26.59%
Drawing or image editing tools (e.g. Inkscape, Gimp, Paint, Illustrator or others.)	14.55%
Hand sketches on a paper	44.32%
Other. Please specify.	16%

Table A.4: Results of Question A.1.4

Other answers: Programming, Mindmaps, Metamodeling with DSMLS, SDMLib, Prototyping in code, Mockups.

A.1.5 What is the typical relationship between your designs and their implementations?

Answers	Response	Percentage
Auto-generate code from design, and after generation, you discard the design.	0	0%
Auto-generate code from design. If design changes, regenerate again.	13	59%
Design is just for documentation and a reference for implementation.	13	59%
Other. Please specify.	3	14%

Table A.5: Results of Question A.1.5

Other answers:

- Code is the design.

- Ecore in EMF and declarative model query definition are both design, always re-generated after change.
- The code embodies the design.

A.1.6 Are you familiar with object-oriented design patterns?

Answers	Count	Percentage
Yes	22	100%
No	0	0%

Table A.6: Results of Question A.1.6

A.1.7 Can you name two object-oriented design patterns from the top of your head without consulting any external source?

Design Pattern 1	Design Pattern 2
model view controller	visitor
Visitor	Template
Flyweight	MVC
Observer	Composite
Strategy pattern	Composite pattern
observer	singleton
Visitor	Interpreter
Bridge	Abstract Factory
Singleton	Decorator
Observer	Strategy
Facade	Composition
singleton	factory
Factory	
facade	observer
Composite	Chain of Responsibility
Strategy	Composite
Visitors	Factories
interpreter	factory
factory	visitor
Singleton	Factory
Observer	State

Table A.7: Results of Question A.1.7

A.1.8 What is the tool or language you use most of the time for creating model transformation?

Answers	Count	Percentage
ATL	8	36%
EMF Inc-Query	1	5%
Epsilon	5	23%
Fujaba	1	5%
GrGen.NET	0	0%
Henshin	2	9%
Moflon	1	5%
MoTif	2	9%
QVT-R	0	0%
QVT-OM	2	9%
Viatra2	1	5%
Other. Please specify.	10	45%

Table A.8: Results of Question A.1.8

Other answers: AToMPM, FunnyQT, VMTL, Rascal, Java, SDMLib, Sigma, e-Motions, UML-RSDS.

A.1.9 What percentage of your software development includes model transformations?

Answers	Min	Max	Average	Responses
Percentage	0%	100%	39.71%	21

Table A.9: Results of Question A.1.9

A.1.10 Do you perform any design activity for planning a strategy to solve the problem before implementing the model transformation?

Other activities specified:

- Walkthrough of desired use cases, concept maps.
- Definition of mapping tables and example input and output model pairs.
- Discussion with colleagues.
- Story Driven Modeling.

Answers	Response	Percentage
Yes, I use generic image editing or drawing tools (such as Paint, Inkscape)	3	14%
Yes, I use hand sketches on a paper.	14	64%
No, I directly start implementing.	4	18%
Yes, other activities. Please specify.	4	18%
No, I just use my mind.	3	14%
Yes, the tool I use provides support for that.	2	9%

Table A.10: Results of Question A.1.10

A.1.11 Why don't you perform any design activity?

This question is only answered by the participants who answered “No” for the previous (A.1.10) question. Other reasons:

Answers	Response	Percentage
No tool for designing model transformations.	2	33%
No generic and common language like UML for model transformations.	0	0%
Not needed.	2	33%
Other. Please specify.	2	33%

Table A.11: Results of Question A.1.11

- I've only written simple transformations.
- Design is done incrementally, with unit testing and refactoring.

A.1.12 Do you think it would be useful to have a language for designing model transformations (analogous to UML for software)?

Answers	Count
Very Useless	1
Useless	1
Somewhat Useless	1
Neutral	4
Somewhat Useful	2
Useful	11
Very Useful	2

Table A.12: Results of Question A.1.12

A.2 Introduction of DelTa

In this part of the survey, we introduced DelTa. We provided all concrete syntax elements and redirected the participants to Ergin and Syriani [36] for further reading, because it was the study that introduced DelTa at the time of the survey.

A.3 Entity Relation Mapping Pattern

In this part, we displayed the complete *Entity Relation Mapping* pattern as described in Section 5.2.1. The name was *Entity Relation Mapping* at the time of the survey. Then, we asked the following questions.

A.3.1 Do you understand this design pattern?

Answers	Count	Percentage
Yes	20	91%
No	2	9%

Table A.13: Results of Question A.3.1

A.3.2 Does the design pattern resemble a structure you have previously used? Please specify.

This is a free-form question and the responses are below.

- This makes me think of the template pattern, ie the source model is a direct template for the resulting transformation output
- Story Driven Modeling
- Yes. Translating a UML model into a Hehnshin graph.
- Yes, simple batch transformation
- The language I use the most is ATL thus, most of the transformation I've written follow this pattern
- Yes - this is normally needed for most of the transformations I have developed in the past.

- This looks like mixing UML activity or state models with ATL
- snapshots, or Graph grammars
- Yes, seems pretty close to most rule/mapping patterns in transformation languages
- Partially. I faced this pattern in some MT using ATL.
- 3 simply answered “Yes.”
- 5 simply answered “No.”

A.3.3 Can you explain it in a few sentences?

This is a free-form question and the responses are below.

- a pattern to map an element in a src metamodel to a target metamodel, including trace information
- This design pattern is saying, if there exists a source entity and a matching target entity, and there is no trace link between them, then create it. Apply this exhaustively. Then, if there exist two source entities sEnt and sEnt2 with a relation between them, and the entities have trace links between them and two target entities that are not connected by a relation then connect them with the relation. Apply this exhaustively and then conclude.
- The design pattern, as described strongly via control flow structures, relates elements of two different tree-like languages in a one-to-one mapping strategy, beginning from the roots proceeding to the child elements.
- First map source elements to target elements, then map source relations to target relations. When done in this order, one can be sure that the target elements corresponding to the start/end element of a source relation already exist.
- This is a task I recently implemented for the Henshin-based implementation of the concrete-syntax MTL I am working on (VMTL). There, LHS, RHS, and NAC patterns are expressed in the concrete syntax of the modeling language (e.g. UML), and are subsequently translated to Henshin rules for execution.

- Simple batch transformations can be designed by exhaustively applying a given rule then continue with the next.
- In ATL, I would use two matched rules where one of the matched rules is also taking care of setting the relations.
- UML state machines and transformation specification using ATL
- When you find a pattern in the source model like the one in the EntityMapping part, generate the corresponding target elements in the target models
- Classes are mapped on tables. Attributes are mapped on columns in the corresponding table.
- This is heterogenous and declarative model transformation where the semantic gap between the source and target metamodels is fairly narrow.
- You are basicalle describing a pattern (a la QVT): relations between properties of elements. This is the bread and butter of model transformations.
- A model representing a class diagram is transformed into a set of tables: classes are transformed into tables, and attributes into columns.
- The way this is done in ATL is very similar but using "lazy rules" instead of scheduled transformation rules. It is clearer in Motif than in ATL, since all rules keep at same level, i.e. they are explicitly scheduled. In ATL this planning is implicitly done by the stack of calls.
- There are constructs to define an ER relation

A.3.4 Why not?

This question was optional and is active according to the "No" result of Question A.3.1. It simply asked the reason for not understanding the design pattern.

- no experience with transformation patterns
- i dont understand the delta concrete syntax specification

A.3.5 Do you see how to implement this design pattern in your favorite language? Are there any complications with this implementation?

Answers	Count	Percentage
Yes	15	68%
No	7	32%

Table A.14: Results of Question A.3.5

The complications that are provided by the participants are below.

- expressing an outplace transformation
- relationMapping most likely, just because it involves a larger number conditions to check.
- I see how to implement it, and there are no complications.
- In ATL, there is the trace model automatically available which helps to set the relations on the target side by referring to the relations on the source side
- Maybe using UML activity models and Foundational UML to execute the transformation?
- No, ETL (ATL, etc.) is a DSL highly tailored for this design pattern.
- But, it does not describe transformations/mappings for the meta-attributes (e.g. the class name or the access public/private options)
- not sure the design pattern is the right "recurring way" to solve similar cases in real settings
- The scheduling part will only apply if the language supports imperative constructs
- 5 simply stated "No complications"

A.3.6 Please rate the following properties of the design pattern.

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Understandable	1	0	5	13	3
Easy to implement in your favorite language	0	1	8	7	6

Table A.15: Results of Question A.3.6

A.4 Fixed-point Iteration Pattern

In this part, we displayed the complete *Fixed-point Iteration* pattern as described in Section 4.3. Then, we asked the following questions.

A.4.1 Do you understand this design pattern?

Answers	Count	Percentage
Yes	19	86%
No	3	14%

Table A.16: Results of Question A.4.1

A.4.2 Can you explain it in a few sentences?

This is a free-form question and the responses are below.

- it is a basic looping structure exiting when a specified condition is reached.
- I think I understand, but I'm fuzzy on it. I'm uncertain exactly how the tags work, or if element1 and element2, for example, are guaranteed to be different elements of different metamodels or just different elements of possibly the same metamodel. I think this is saying, set the mark tag on all elements of type element1 and of type element2. Then mark a fixed point type element. If you can't mark it, then randomly either mark an elementToModify type element (subelement of element1 or element2?) with a modify tag (can the modify tag be set repeatedly?), or if anElement exists and there is a relationship to elementToDelete with a tag

”marked” then delete elementToDelete, or if anElement exists and has a marked tag and isn’t connected to elementToCreate then create elementToCreate.

- This pattern is used when a transformation has to do a certain operation until some condition has been satisfied. In the example, the operation is creating links to ancestors by following associations, and the end condition is when a least common ancestor is found (i.e., both nodes link to the same ancestor node).
- Iteratively apply a rule (or a set of rules) until some condition holds.
- Fixed Point Iteration can be used to recursively add, delete, or modify model elements. It is, in a sense, a surrogate for a loop construct. Related to the DelTa implementation, I don’t understand why the deleted element is marked in the Delete pattern, whereas in the Create pattern the created element is not marked.
- you build the result value by successively approximating it in a lattice structure.
- Provide a rule that decides whether a goal has been met. Keep applying a rule that performs some kind of marking or exploration of the model. After each application, check the goal.
- It’s behaviour is analogous to a ‘while’ loop in a general-purpose language.
- Rewrite a model with the same rule applied for different matches until there is a condition fulfilled/not fulfilled.
- Get next ancestor of both elements in each iteration until a common ancestor is found or no more ancestors are present.
- It handles a loop until a specific condition is satisfied.
- add new ancestors to A and B until a common one is found
- This is an in-place model transformation that uses a fixed state of the model as the stopping point.

- The fixpoint is a guard to test the model changes. You either modify, delete or create elements until the guard is met.
- An iterative process, where elements are marked and processed until a fixed point is reached
- This MT tries to link all nodes to its antecesor.

A.4.3 Why not?

This question was optional and is active according to the “No” result of Question A.4.1. It simply asked the reason for not understanding the design pattern.

- I cannot map the object identifiers from initiate to checkFixedPoint and cannot trace on which elements the pattern is operating. I mean where is element1 and element2 in checkFixedPoint and where do elementToModify and anElement come from? If this is the correct syntax, the specification has too much ”magic” for my taste.
- The attributes in the boxes re not clear. The relationship between the elements in the different phses are not clear.

A.4.4 Do you see how to implement this design pattern in your favorite language? Are there any complications with this implementation?

Answers	Count	Percentage
Yes	15	68%
No	7	32%

Table A.17: Results of Question A.4.4

The complications that are provided by the participants are below.

- If my understanding is correct, it is mostly straight forward. Perhaps checking the conditions for the create type.
- With Story Driven Modeling, and it would seem very similar to your pattern picture, whereas I would be careful with object identifiers

- I see how to implement it and there are no complications.
- No complications, though I suppose modification, creation, and deletion would end up implemented as separate transformations.
- No complications
- The implementation is not trivial but it's possible
- Having a language which allows to orchestrate rules and to define conditions on the model should enable to implement the pattern such as in Henshin.
- UML already provides the possibility to create loops
- EPL is a domain-specific language tailored to this style of transformation.
- These iterative processes are hard to implement in ATL. In the figure, it is hard to determine the type of elements to process. I consider the diagram confusing (e.g. the "fail" in the fixed point to continue, or the different look of the block grouping rules)
- again, this looks like an overkill for the purpose
- I think there are no explicit links that can be set in Henshin, must be a property in mm
- Needs inplace semantics
- easy
- You are assuming that the language supports imperative constructs. Additionally, it may be the case that due to dependencies the fixedPoint may never be reached (i.e. another rule or loop needs to be executed first, at the same time, etc.)

A.4.5 Please rate the following properties of the design pattern.

The following explanations were added by participants.

Question	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
Understandable	1	2	8	9	2
Easy to implement in your favorite language	2	3	4	11	2

Table A.18: Results of Question A.4.5

- Not easy to understand (reasons given above); not all MT languages implement in-place semantics
- It is confusing.

A.5 Finalizing The Survey

We asked the following question to summarize the survey.

A.5.1 Is it appropriate to design the solution using a specific notation first, before implementing the transformation?

Answers	Count	Percentage
Yes	18	82%
No	4	18%

Table A.19: Results of Question A.5.1

Below are the reasons behind the answers.

- A specific notation is not necessary. However, design is a crucial step and the notation used should facilitate all relevant aspects of system design.
- If the notation is helpful in visualizing and structuring the transformation, then it would likely help produce an easier to follow transformation ruleset.
- I think it only makes sense to design model transformations, as they are 'programs' too. The implementation (i.e., the rules and schedule) is often very close to a 'design' already, however. In this sense, a model transformation can already be seen as a (executable) program design. The added value of the design patterns presented here is, it seems, purely for documentation purposes and a transposition

of those presented in “The Gang of Four” book. The difficult part, of course, is recognizing when a specific design pattern is applicable, and then implementing it.

- If you mean a ”standardized and commonly used notation” by saying specific notation, then no. I would just choose a transformation language which provides me a useful notation for the specific task. I would ask myself questions such as: ”Do I need control flow structures? Do I deal too much with traceability links? ...etc.” Then my chosen transformation language would have the proper notation.
- At least for the two patterns presented in this survey, any implementation of these patterns in a concrete scenario (i.e., with a concrete transformation language and concrete metamodels) will look pretty similar. So I don’t see a benefit in specifying the transformation first in yet another language.

With UML, one major benefit is that you can illustrate, e.g., the structure of a subsystem and the associations between classes very concisely. Illustrating the same using Java code with only classes with attributes is not as concise and clear. However, to me the DelTa specs look just as concise and clear as a concrete implementation in some transformation language.

Of course, if I could use DelTa to specify a transformation and then it spits out a set of concrete transformations in different transformation languages, that would be a benefit. Then I could just go with DelTa and use the generated transformation which fits my use-case best, e.g., the fastest one or the one which is accompanied by the most comprehensive toolset.

DelTa might also be useful to reason about transformations without having to know each and every transformation language in detail. Therefore, some DelTa reverse engineering facility would also be interesting. That would read in a transformation in a concrete language and present it using DelTa. However, that’s probably not very feasibly because most realistic transformations cannot be expressed using high-level concepts such as element-mapping only but they frequently use tool-specific low-level stuff, e.g., for the computation of attribute values.”

- The design phase (at least in its early stages) is meant to capture the transformation's intention and abstract away from any implementation details. This phase is inevitable, but is often invisible, as it occurs in the transformation developer's mind. Making it explicit would benefit large transformations and transformations involving several developers - but is probably of limited usefulness for small transformations.
- It may help documenting knowledge in some area or domain.
- Partially, yes. However, I think that the design will not go down to the element level modifications. Designing the overall flow of the transformation and break it up to phases and rules with preconditions identified is useful.
- The design phase is very important in every software development process. Although MDE is still an immature field in comparison with OOP, structured programming, etc., as soon as model transformations gain popularity, the design phase will become more and more important.
- As the design should explain more about what has to be achieved, the implementation has to provide the actual transformation logic. Maybe there are different possibilities to implement a specific pattern w.r.t. performance, scalability, memory consumption, and so on. Furthermore it allows to document the rationale behind a model transformation implementation which is very hard to reverse engineer as model transformations tend to get complex if they grow in size.
- Generic language useful for communication. However translation into concrete languages might prove difficult.
- beneficial for testing
- Sketching the transformation first helps to understand if every part of the source element can be transformed into a part of the target element – finding transformation limitations

- abstraction is a good mechanism for coping with complexity.
- Easy to read and thus helpful for design
- It really depends on context! Sometimes, I think it is extremely valuable to spend a lot of time on design (e.g., if this is to be a long-lived transformation that we will maintain for many years) and sometimes I think it is best to spend almost no time on design (e.g., if the transformation is just to quickly explore an idea).
- Yes. It will help you understand the complexity of the relations between your candidate models and the attributes you want to map.
- An standard notation in the correct abstraction level help designers/developers to reason about the solution of complex problems. However, if the modeling is more complex than build the transformation, the modeling is not useful.
- I'm convinced there can be cases where a specific notation is going to be useful, but just like design patterns, a systematic usage of them will make an implementation overly complex for no gain at the end.
- Because you can reason about the solution. Furthermore, you can discuss the propose solution with other colleagues.
- It helps to refine the idea

A.5.2 Do you think it would be useful to have the DelTa as a language for designing model transformations (analogous to UML for software)?

Answers	Count
Very Useless	0
Useless	1
Somewhat Useless	2
Neutral	7
Somewhat Useful	6
Useful	6
Very Useful	0

Table A.20: Results of Question A.5.2

Below are the additional comments from participants.

- I do not see the added value compared to programmed graph transformations (lets say Story Driven Modeling) which has the very similar notation and serves as an implementation anyway.
- I'm not convinced that yet another modeling language (for design patterns) will help here.
- If the language is specifically tailored for designing model transformations, it is more useful than doing the same in a generic image editor (e.g. visio)
- Model transformations are complex artefacts which may greatly benefit from design descriptions and the patterns and how they are implemented are important ingredients for transformation designers - especially for beginners.
- Easy to understand + graphical notation.
- language should be explained in more detail before presenting examples
- Languages like ATL are textual and quite difficult to manage for many users, such a graphical notation could lower the learning curve
- Again, abstraction permits dealing wit complexity
- It should have code generation or an interpreter.
- I don't know how to answer this. I would have to use DelTa on several projects to be able to form a useful opinion.
- 1. It assumes the langauge has imperative constructs. 2. It just seems a visual representation of the transforamtion rules. It does not provide any abstraction or additional information (i.e. dependencies). A class diagram lets you rapidly identify dependencies, inheritance issues, poissible generaliations and abstractions, candidate interfaces. I don't see how DelTa can provide such helpers which is what would really mean designing the transformation.

- I think that complex transformations will be hard to model and simple transformations may be do not need modeling.
- Would need to know more about DelTa

APPENDIX B

DELTA GRAPHICAL CONCRETE SYNTAX

In this appendix, we present the graphical concrete syntax of DelTa. All elements that are already explained in Section 3.3.2 on Figure 3.2 are depicted in Figure B.1 to Figure B.5 individually.

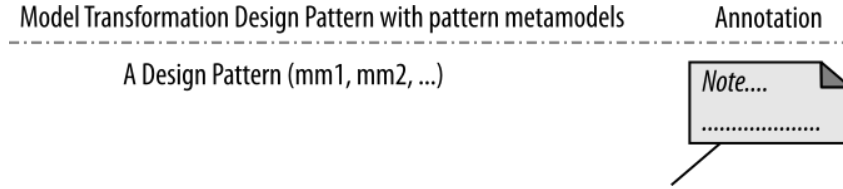


Figure B.1: Model Transformation Design Pattern and Annotation

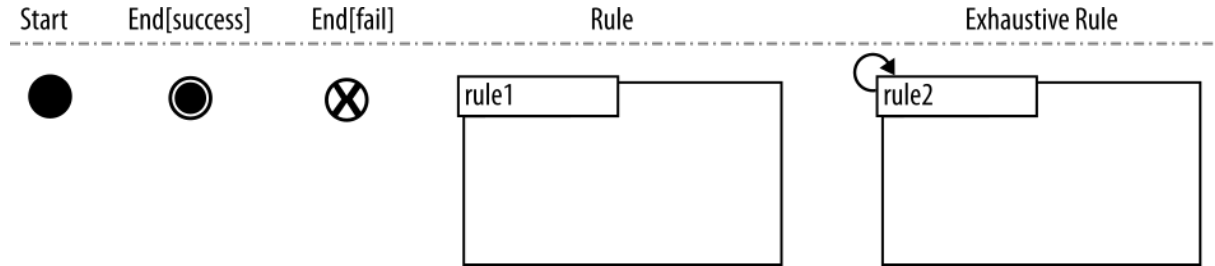


Figure B.2: Transformation Units

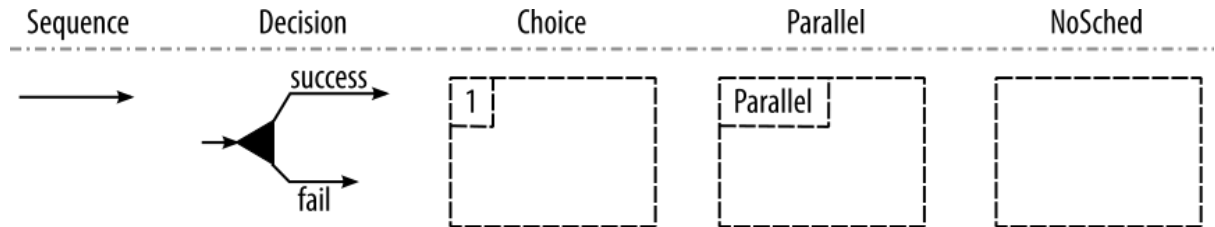


Figure B.3: Transformation Unit Relations

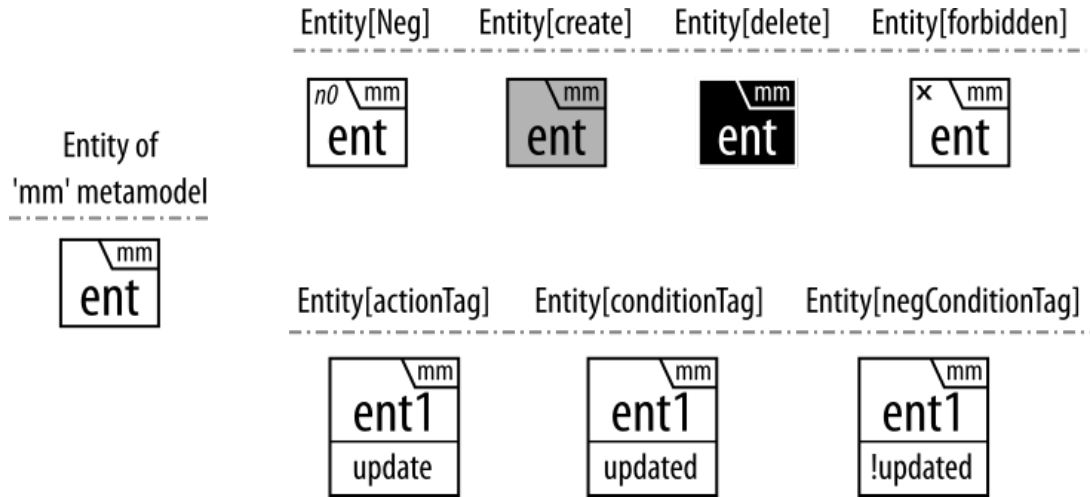


Figure B.4: Pattern Metamodel - Entity

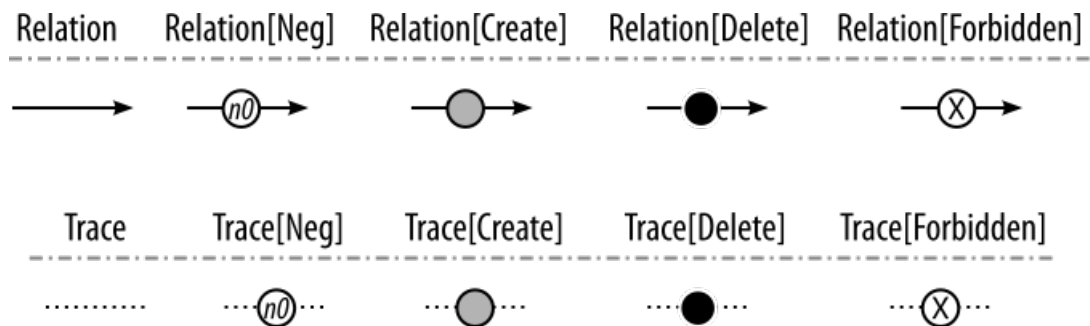


Figure B.5: Pattern Metamodel - Relation and Trace

APPENDIX C

USER STUDY OF THE METHODOLOGY

This appendix has all the questions and results of the survey of the user study mentioned in Chapter 7 to validate the methodology. We used Qualtrics to distribute the survey and collect the results. In the free-form questions, all grammatical errors of the participants are left as is for authenticity.

C.1 Have you ever used GrGen.NET model transformation language before?

Answers	Count	Percentage
Yes	1	10%
No	9	90%

Table C.1: Results of Question C.1

C.2 Did the methodology (the 5-step design pattern driven approach) have any impact on your way of thinking the solution for the problem?

Answers	Count	Percentage
Yes	7	70%
No	3	30%

Table C.2: Results of Question C.2

Justification of their answers are below in free-form text.

- Tried to see how to solve the problem with the given pattern instances. I would have used an entirely in-place solution only with retyping without the given patterns.

- It provides a way to visualize the potential solutions. It easier to solve the problems since the tool provides with an abstract of the transformation. I think it definitely traces the path to the correct transformation.
- I decided to explore the available gallery of transformation patterns rather than directly implementing a solution.
- the method helps to think about the way forward to solve the problem
- The methodology helps to structure how solve the problem, but does not guarantee to reach a solution.
- The methodology helped me to think on how I can design the transformation I want to make, before I start to implement the transformation.
- It made me step back and think in terms of the provided design patterns rather than going off on my own.

C.3 Please give us your appreciation on the design pattern driven approach.

Questions	Excellent	Good	Average	Poor	Bad	Responses
Did you understand it?	4	5	1	0	0	10
Is it useful?	3	5	2	0	0	10
Do you find it natural?	0	5	4	1	0	10
Would you adopt it in the future?	2	5	2	0	1	10

Table C.3: Results of Question C.3

Justification of their answers are below in free-form text.

- Did not understand the Visitor pattern; what it is good for? Some prose about the intent would be helpful, instead of just a structural image. Did not expect the trace stuff getting generated for the transitive closure pattern. (Communicating knowledge (derived from experience) vs generating skeleton for novice users to extend vs some code generator for comon idioms saving boilerplate coding)

- 3 excellent because it helped minimizing the effort of writing all the transformation.
Good for natural
- I mostly understood the design patterns, though the visual representation is not the most intuitive. Reading the description was necessary to grasp what each pattern was doing, though this button was not explained in the tutorial. Using patterns seems like a natural way of doing this, but it is, in this approach, difficult to combine several patterns together: the tool generates the files for a single pattern, but combining them is still up to the user in the usual textual environment. Because names of the entities cannot be chosen manually (or given shorthand names), it becomes more difficult to further refine the rule, such as writing the “eval” part: you need to look up what is the name of the entity you want to modify (e.g., sEnt). More familiarity with the different kinds of patterns would be required for me to adopt this approach in the future: now it is time consuming to look through all the patterns that exist (reading the description for each of them). Maybe they could be categorized?
- I believe that beginning with a design philosophy that encourages the use of existing well-studied common patterns to solving problems is a good practice. However, using a tool to automatically generate partial solutions that are then manually modified to properly solve the intended problem is not the most natural process. Typical development works from scratch and progresses directly toward the solution with all components being added directly addressing the intended problem. The refactoring stage is the unnatural portion for me. Furthermore, the added difficulty of beginning with a partial incorrect solution when I could directly implement the structure intended would decrease the likelihood of me following this specific methodology as presented here. I would (and often do) use the first few steps to identify relevant patterns from existing well studied patterns or patterns that I have personal experience with from past projects (i.e., drawing out small segments of functionality from existing projects).

- the greatest impact on the way of thinking was the transformation language itself, I understood the approach, but I found the customization limited leaving the patterns quite dissociated from what I was searching, and with the limited library of patterns, its hard to establish if with a full library it would be better I didn't find it completely natural, as I would think of the patterns, but it forces me to stop too much before implementing them. probably with some training I might adopt this type of approach to better define the patterns I use.
- its helpful to have design pattern to design transformation model but need good knowledge of patterns
- The approach is interesting, but learning time may be longer than the time for this study. However it seems useful.
- There was a lot of text, which made it take it a little while to understand (inline graphics for each number might make it a little easier to consume). I think the design pattern concept though is very effective.

C.4 Please rate the language DelTa that defines the structure of design patterns.

Questions	Excellent	Good	Average	Poor	Bad	Responses
Understandability of design patterns	2	4	3	1	0	10
Readability of design patterns (concrete syntax, shapes etc.)	4	4	1	1	0	10
Usefulness	3	4	3	1	0	10
Appropriateness	2	7	1	0	1	10
Completeness	1	5	3	1	1	10

Table C.4: Results of Question C.4

Justification of their answers are below in free-form text.

- The intent of patterns must be explained, just giving a diagram is not sufficient (regarding understandability). The patterns are built against a certain lowest common denominator of graph rewriting/model transformation. Some tasks will be overly

complex when they are applied, instead of tool specific more powerful constructs. Given the explanations of the notation, it s quite clear whats going on (without them: no). What are the benefits of DelTa compared to using e.g. story diagrams?

- The design patterns can be understood easily. However, the link between the graphical part and the parameters is not straight forward.
- I know DelTa, but i'm not an expert
- The design patterns themselves are not completely readable with the purely graphical syntax. Some kind of comment (or description) would be very useful to understand what the pattern actually tries to do. I was unfamiliar with the concrete syntax, and don't know why no more intuitive syntax was used, for example using colors. Furthermore, the way a NAC is defined seems strange to me. I understand that you place a "n" followed by a number on the association, but does this also scale to entities that should not exist? I do believe that the language is appropriate and seems to be complete.
- I feel that with more exposure to the syntax and semantics of this language that it would be perfectly acceptable. However, after the minimal exposure provided here, I was still struggling to understand exactly what the presented design patterns were doing. Without the text descriptions, I would not have been able to decipher the patterns.
- its usefulness is very much tied to its understand-ability, with training it will become more useful, the design patterns themselves are clear and mostly readable, there are some visual conflicts between the arrows inside a rule and between rules. its very similar to statemachines and that is appropriate, as it provides a base of understand-ability, as for completeness, there are some situations of conditional application that are not covered, but that was also hard to describe on grg and other transformation languages
- DetTa looks complete for these patterns.

- The color in the shapes is not entirely clear (When black for example) and it may lead to confusion.
- I ran into a few issues, such as not being able to add multiple edge types in the NAC. Additionally, testing properties felt a little unintuitive. Otherwise, this feels like a good, concise language. Though again, edge syntax feels a little odd, I would have expected something like $elem1 - > edgeType1 - > elem2$ for unnamed edges and $elem1 - > e1 : edgeType1 - > elem2$ for an edge that you want to assign to the var `e1`.

C.5 Please rate the tool that generates partial model transformations

Questions	Excellent	Good	Average	Poor	Bad	Responses
Easiness to use	4	4	2	0	0	10
Intuitiveness	3	5	2	0	0	10
Usefulness	5	3	2	0	0	10
Correctness	5	5	0	0	0	10

Table C.5: Results of Question C.5

Justification of their answers are below in free-form text.

- The tool does what it should do regarding the test. Outside this limited frame I think it will work, but what are the benefits that make it worthwhile? Assuming it offers many more patterns: The tool generates some GrGen concrete syntax from some placeholders getting filled in. The value for an expert GrGen user is in the amount of typing saved, compared to doing it manually. For a novice its the skeleton generated. A catalog of patterns communicating knowledge is then again sth of value – but that requires a full catalog.
- It would be easier if the tool could already present some suggestions as to which entities can be selected. For example, as soon as the metamodel is selected, it should be able to give you a list of entities that are applicable for a specific part of the pattern. This would make it easier to know what you can do, and also prevents

copy/paste errors. The generated patterns seem to be correct, though it is difficult to understand the generated patterns, which is (later on) required to further refine or chain them together. Generation of some comments that would help can make this nicer. Optionally, maybe there should be a generic box present in the pattern, where you can put in your eval statements.

- I would have liked to have the auto-generation be able to generate the rules within the existing transformation thereby removing the need to manually copy over code segments. Also, I understand the desire to work with a standalone tool and not have to manage the dependencies or issues of an existing environment, but it would be nice to see the solution integrated within an environment such as Eclipse (could target a variety of transformation languages and modeling tools provided by Eclipse). This would also facilitate integrating generated segments into existing transformations, and it would keep developers within the development environment context without needing to jump out and use a separate tool.
- when filling out the template fields it might be helpful to highlight the pattern element being customized due to the uniform nature of the patterns it is not always clear where in the pattern we are customizing and the implications on the generated code
- Maybe parameters to fill should be more clear, like something to indicate the type of fields (for example, `createNewEntity` is the name of the rule)
- In general, the approach is very interesting, The methodology seems very clear and in the future can be very useful. The directives can be a bit long but are very detailed.
- It would be better to add documentation on each design pattern to help the user
- Once I got how elements link to the MM, it made more sense, but it took me a little while to understand how the pieces clicked together. It's natural to think in

terms of an overall pattern, though I think a more dynamic interface would help more (ie, change labels on the model itself)