Design Pattern Oriented Development of Model Transformations

Huseyin Ergin[a], Eugene Syriani[b], Jeff Gray[a]

[a] *University of Alabama, Tuscaloosa, Alabama, U.S.A.*
[b] *University of Montreal, Montreal, Canada*

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. 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. Given their various contexts of application, model transformations may also benefit from design patterns. Although several studies have proposed design patterns for model transformation, there is still no accepted common language to express transformation patterns. Therefore, we propose a semi-formal way to describe model transformation design patterns that is independent from a specific transformation language and described in a practical way that is directly implementable by model engineers. This paper presents a catalog of 15 model transformation design patterns. We also demonstrate how it is possible to automatically generate excerpts of a model transformation in various languages given a design pattern. We conducted an initial survey to motivate the need for model transformation design patterns and a user study to validate the methodology we propose to solve problems as model transformations based on design patterns.

Email addresses: hergin@crimson.ua.edu (Huseyin Ergin), syriani@iro.umontreal.ca (Eugene Syriani), gray@cs.ua.edu (Jeff Gray)

Keywords: design patterns, model transformation, model-driven engineering, development process, DelTa

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 [1]. MDE uses models to describe complex systems at multiple levels of abstraction. 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 which 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) [2].

MDE developers use model transformations for various activities, such as evolving, refactoring, simulating, and manipulating models [3]. These are supported by a plethora of model transformation languages (MTLs) [4], such as GrGen.NET [5], Henshin [6], and MoTif [7], just to name a few. 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 [8]. As witnessed in [9], 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.

One of the most revolutionary contributions to software design was the Gang of Four (GoF) catalog of object-oriented design patterns [10]: both in designing the software before implementation and automatically generating code from the design. Similarly, we believe that the design of model transformations can benefit from model transformation design patterns. 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"* [10]. This definition has been adapted for graph transformation [11] and, more recently, generalized for model transformation [12]. Design patterns are used in a multitude of software engineering areas, such as in parallel programming [13], finite-state verification [14], but also in other aspects of MDE, namely for domain-specific languages (DSLs) [15] and metamodels [16]. 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 [18], given they provide a common vocabulary for design [19].

Several design patterns studies have been proposed for model transformation [11, 12, 20, 21]. 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 UML diagrams in order to make the design pattern structure more readable and understandable, which also greatly helps the automatic generation of the software code [22]. As stated in [23], 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 engineer[1]. Additionally, a pattern language must allow one to verify if a transformation correctly implements a pattern. Design pattern catalogs evolve over time and new patterns keep on being discovered due to the evolving nature of software and reuse habits of model engineers [24, 25, 26]. 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 language can be found in [27]. More recently, Lano et al. [12] published a broader study about the topic.

In this paper, we explore existing studies in model transformation design patterns and identify 14 unique "real" design patterns in the literature. The first contribution is a new unified formalism to describe model transformation design patterns that consists of a template to describe and discuss a pattern along with a modeling language to represent the structure of its solution. The second contribution is the conduction of a survey across the community of model transformation engineers to identify the needs for design patterns and a dedicated unified language to express them. This paper can be consid-

[1]Through the paper, we use "model engineer" in place of "user" or "developer" of a model transformation.

ered a response to the demands of the model transformation community in matters of design patterns. The third contribution is a process, aided with a tool we implemented, to guide model transformation engineers in their design by automatically instantiating patterns in the MTL of their choice, using template-based code generation. Finally, we define an additional design pattern as the 15th, "execution by translation."

The rest of the paper is organized as follows. In Section 2, we present the results of the pilot survey, where we motivate the need for a language to express model transformation design patterns. In Section 3, we explain the terminology to distinguish between different reuse structures and discuss existing work, the different notations used to express model transformation design patterns, along with the classification of the patterns in the literature. In Section 4, we present the unified template to describe model transformation design patterns and, in particular, the modeling language DelTa to define the structure of the patterns. In Section 5, we demonstrate how all existing model transformation design patterns, as well as new ones, use the unified template. In Section 6, we propose a development process of model transformations driven by design patterns. We also discuss the implementation of the model transformation generator. In Section 7, we validate the methodology along with the tool and the language with a user study. Finally, we conclude in Section 8.

2. Pilot Motivation Survey

In order to determine whether a design pattern language for model transformation is useful, we conducted a survey. The research questions identified are:

RQ1 Is there a need for a common language to describe model transformation design patterns?

RQ2 Is DelTa an appropriate candidate to describe model transformation design patterns?

RQ3 How can a model transformation design pattern improve the implementation of model transformations?

2.1. Data Collection

We have prepared an online survey[2] with a total of 22 questions. We have also supported some of the questions by asking the reason behind the answer. The survey was closed to selected participants only. We have used the Qualtrics[3] software to analyze the results.

2.2. Experimental Setup

There was no time limit to complete the survey and participants had access to any resource they needed. The survey consists of four blocks of questions or explanations. The first block has 10 questions and focuses on background information about the participants, such as familiarity with design patterns, software design, and model transformation. The second block has no questions but introduces DelTa and its purpose in a paragraph along with referring the participant to another document that shows DelTa concrete syntax in details. In the third block, we test 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 [27]. Here, the level of complexity is relative to the number of constructs used in the design pattern. We asked 4 questions for each of the design patterns. The final block has 4 questions and collects their opinion regarding the three research questions.

2.3. 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 Transformation, Transformation Tool Contest). Additionally, the participants must have developed at least one model transformation in the past. A total of 23 participants ended up completing the survey. According to the background question results, 95% of the participants develop software for academic purposes and an average of 27% of their development time focuses on the design phase. 44% of the participants use hand sketches for designing and 26% use 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.

[2] `http://tinyurl.com/DelTaSurvey2015`

[3] `www.qualtrics.com/`

Language	Used by
ATL [28]	36%
ETL [29]	23%
Henshin [6]	9%
MoTif [7]	9%
QVT-OM [30]	9%

Table 1: The five most used model transformation languages (multiple choice)

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

2.4. Results of Transformation Survey

Table 1 shows the most popular model transformation languages among the participants. They could choose multiple selections from 11 languages we proposed and another field where they can enter the name of an MTL not listed. Table 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. Some languages (e.g., MoTif, GrGen.NET) have dedicated IDEs that let the model engineers design immediately. Table 3 depicts the participants' understanding of the design pattern and how to implement it in their language. Finally, 82% of the participants agree that it is appropriate to design the solution using a specific notation first, before implementing the transformation and 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 available online[4].

[4] `http://tinyurl.com/DelTaSurveyResults`

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

Table 3: Comprehension of the design patterns

2.5. 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. 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-scale, from strongly agree to strongly disagree. The results are in Table 3. Nevertheless, we did not have empirical evidence of this when we conducted the survey, but now Section 6 validates this statement. It is important to note that the survey used the earlier version of DelTa presented in [31], 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. This led to the version of DelTa presented in Section 4.2. 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 MTLs. However, DelTa is not an MTL, but a language for describing design patterns which abstracts concepts present in existing MTLs. The other participant was worried that DelTa could 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 MTL should definitely help. In addition, model engineers think it may help to document the knowledge in the domain and understand the complexity of the transformation before implementation.

2.6. 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 [27] and a reference guide to understand graphical syntax of DelTa as depicted in Appendix B. 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. One other threat is the number of participants and how far we can generalize the results. We have asked questions about their experiences with model transformations.

3. Existing Work on Model Transformation Design Patterns

In the following, after we define the terminology, we discuss initial works on design patterns for model transformation and, in particular, a recently published catalog [12]. We also classify the existing design patterns according to our terminology.

3.1. Terminology

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 model transformation language, which 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 [11]. They are often presented as a feature built into the MTL, e.g., *multiple matching pattern* in ATL [21].
- **Design patterns** are language-independent solutions to a class of problems and are applicable to any MTL [10], e.g., *mapping pattern* [20].
- **Refactoring patterns** are language-independent structures that improve an existing transformation according to some quality criteria [32], e.g., *introduce rule inheritance* [12].

3.2. Reusable Idioms

Initial studies on model transformation design patterns proposed useful idioms that are specific to model transformation languages: GReAT [11], QVT-R [20], ATL [21], and VMTS [33]. 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 design patterns for model transformation was by Agrawal et al. [11]. 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 patterns are defined in the GReAT language.

Iacob et al. [20] defined five other design patterns for outplace transformations, where the input model is not modified during the transformation and the output is a separate model. The *mapping* pattern 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 [27]. The *refinement* pattern 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* pattern abstracts a specific type of node from the target model while preserving its original relations. The *flattening* pattern removes the composition hierarchy of a model by replacing the containment relations. The *duality* pattern 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. [21] mined ATL transformations and discovered two design patterns. The *transformation parameters* pattern 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* pattern 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 pattern obsolete.

Levendovszky et al. [33] proposed domain-specific design patterns for model transformation as well as different DSLs. In their approach, they defined design patterns using a specific MTL, VMTS, where rules support metamodel-based pattern matching. They proposed two design patterns: the *helper constructs in rewriting rules* pattern explicitly produces traceability links, and the *optimized transitive closure* pattern which is similar to the one from Agrawal et al. [11].

3.3. Design and Refactoring Patterns

Recently, Lano et al. [12] 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." [12] These include the *phased construction* pattern to decompose a transformation into phases. Optimization patterns are "concerned with improving the efficiency of a transformation." [12] 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." [12] 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 [3].

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.

3.4. Classification of Existing Efforts

In Table 4, we classify existing pattern structures in model transformations according to the terminology we have provided in Section 3.1. Design patterns marked with '*' are classified as reusable idioms, but they can be promoted to design patterns if more MTLs support the feature they depend on.

Most patterns from the *rule modularization* category are considered design patterns according to the above definitions. *Phased construction* is a general pattern that encompasses *structure preservation*, *entity splitting*, *entity merging* and *map objects before links*. 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. This is identical to the *Entity Relationship Mapping* pattern that we published in [27]. *Auxiliary metamodel* is a pattern to support temporary elements that do not belong to source or target metamodels. *Construction and cleanup* again show some similarity to *Phased construction*, since the construction phase is clearly separated from the cleanup phase. *Parallel/serial composition* 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* provides a for-all support by using double-negations and simulating explicit rule scheduling for languages that have implicit scheduling.

Study	Design Pattern	Refactoring Pattern	Reusable Idiom
Lano [12]	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 [21]	Transformation parameters[b]		Multiple matching
Levendovsky [33]	Optimized transitive closure[d], Helper constructs in rewriting rules*		
Agrawal [11]	Transitive closure[d], Leaf collector[c]		Proxy generator
Iacob [20]	Mapping[a], Node abstraction*	Refinement, Flattening	Duality
Ergin [27]	Entity-relation Mapping[a], Transitive Closure[d], Visitor[c], Fixed-point iteration		

Table 4: Classification of model transformation design patterns. Same patterns with different names are annotated with same letters (e.g., Model visitor and Leaf collector).

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

We have generalized the patterns *recursive descent* and *model visitor* to the *visitor* pattern [27]. They work on hierarchical structures and require processing of nodes in this structure.

Architectural patterns are generalizations of rule-level patterns to organize transformations. In Lano et al., the structure is provided using activity diagrams. Although they 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 in this study 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 [7]. 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.[12] 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., depicts how not to do things), such as entity merging and entity splitting.

Bezivin [21]'s *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 [33] is a design pattern that can be identified in some other studies [11, 27] and also in this paper. *Helper constructs in rewriting rules* is considered a design pattern, because creating traceability links can be reused in various MTLs.

Agrawal et al.'s [11] *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.

We generalize Iacob et al.'s [20] *mapping* pattern in this paper to an *Entities before relations* pattern. *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. [27] are design patterns designed for the sake of this study.

4. A Unified Template for Model Transformation Design Patterns

This section defines the template to use when describing a MTDP.

4.1. The Unified Template

According to the feedback gathered in the survey in Section 2, although DelTa is a good candidate to describe a design pattern, it is not sufficient alone. A more complete description similar to GoF [10] design patterns was suggested. As shown in Section 3, there is no agreement on how to represent model transformation design patterns. Different studies have used different fields to represent a design pattern, e.g., applicability, benefits, and structure. Table 5 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 5 show which fields 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 fields as a unified template shown in the first column. They are mostly influenced by Lano et al. [12] since it was the most complete and thorough template in the literature. In the unified template, a design pattern consists of the following fields:

- **Summary:** a short description of the design pattern that usually gives the outline of the other fields in a few sentences.
- **Application Conditions:** pre-conditions on the context of use of the pattern. The conditions can be either pre-conditions on the metamodel or constraints over the transformation. This is usually expressed in the same language as the solution field.
- **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 again 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 [21]**	**Levendovsky [33]**	**Agrawal [11]**	**Iacob [20]**	**Lano [12]**	**Ergin [34]**	**GoF Meaning**
Summary	Motivation	Motivation	Motivation	Goal	Summary	Motivation	Intent
				Motivation			Motivation
Application Condition		Applicability	Applicability	Applicability	Application Conditions	Applicability	Applicability
Solution	Solution	Structure	Structure	Specification	Solution	Structure	Structure
							Participants
Benefits	Consequences	Consequences	Benefits		Benefits		Consequences
Disadvantages			Limitations		Disadvantages		
Examples		Known Uses	Known Uses	Example	Application and Examples	Examples	Known Uses
							Sample Code
Implementation						Implementation	Implementation
Variations		Variations	Variations			Variations	
Related Patterns					Related Patterns		Related Patterns

Table 5: Comparison of fields for design pattern description

4.2. Design Pattern Language for Model Transformations

Below, we define the language we have created , DelTa, to express the solution field 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. [10] 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.

4.2.1. Abstract Syntax

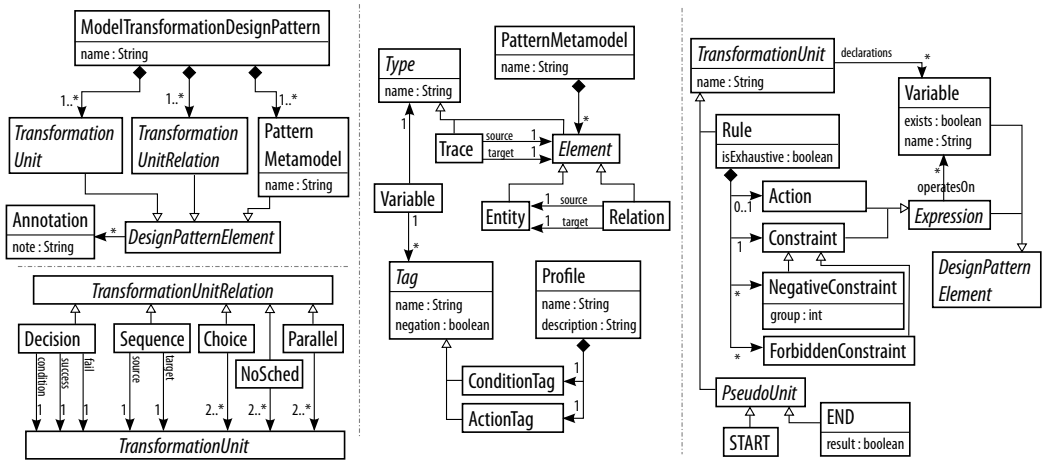

Figure 1: DelTa Metamodel

As depicted in Fig. 1, a model transformation design pattern (MTDP) consists of three kinds of components: transformation units (TU), pattern metamodel (PM) and transformation unit relations (TUR). This is consistent with the structure of common MTLs [35]. In MTDP, rules represent a similar concept to graph transformation rules [36]. A rule consists of a constraint, an action, optional negative constraints, and forbidden constraints. The first

three correspond to the usual left-hand side (LHS), right-hand side (RHS) and negative application conditions (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 pattern metamodel or a trace. The PM is a label to distinguish between elements from different metamodels, since a MTDP is independent from the source and target metamodels used by the concrete model transformation. When implementing 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 [37]. 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 implementing 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 [35], 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 [6] 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 [35].

As surveyed in [7, 4], in any MTL, rules are subject to a scheduling policy, whether it is implicit or explicit. For example, AGG [38] uses layers, MoTif and VMTS [39] use a control flow language, and GReAT defines causality relations between rules. As shown in [40], 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 4.2.3. 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.

4.2.2. Concrete Syntax

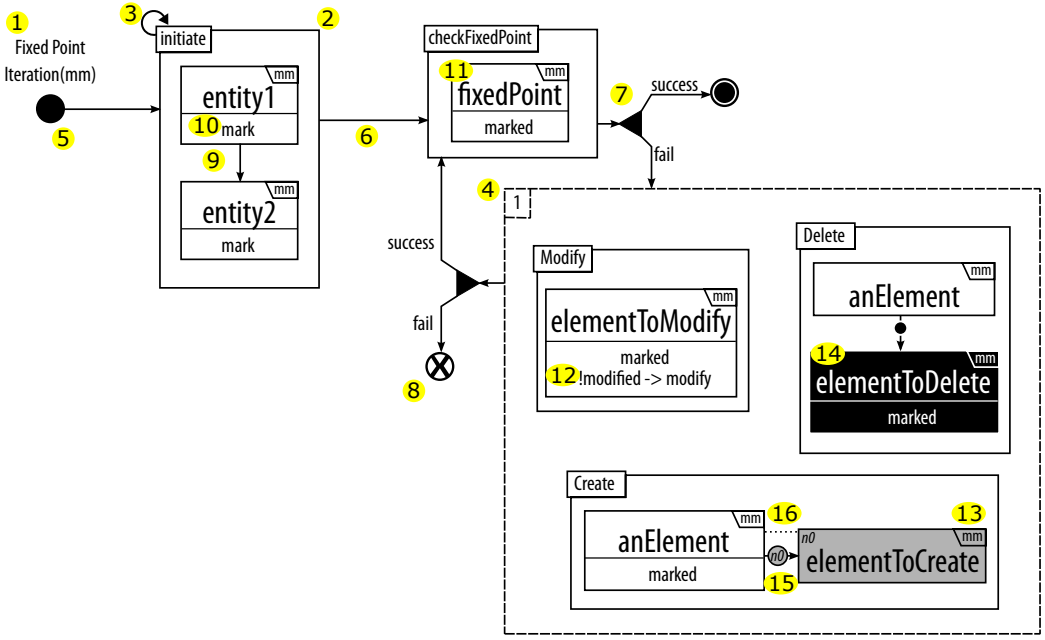

Figure 2: A Sample Pattern in DelTa Concrete Syntax

We highlight the DelTa graphical concrete syntax through an example in Fig. 2. A textual concrete syntax is also available in Appendix A. The

figure depicts the structure of the solution of a sample pattern we have modified from Fixed-point iteration pattern to illustrate most of the elements. Therefore, it is not a real design pattern.

1. A design pattern has a name and takes as 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 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 predence 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 one from the choice block.
8. The design pattern can also state that the concrete transformation implementing it should terminate unsucessfully. 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 be 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 **elementToModify** 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 negative application condition (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.

4.2.3. Informal Semantics

The semantics of MTDP rules is borrowed from graph transformation rules [36], 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: $constraint \wedge \neg neg1 \wedge \neg neg2 \wedge \ldots \rightarrow action$. 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. Since 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 [5]. 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 [7] and GReAT [41], and parameter passing in Viatra [42]. A condition tag should be used as a verb in past tense form and an action tag should be used in imperative form. In the 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 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 START and end with a number of ENDs. 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.

4.2.4. Comparison of DelTa with Existing Languages to Express Design Patterns

Guerra et al. [9] 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 (RD), which represent the language that automatically produces the implementation of the transformation, are used to describe the structures of the rules and their task in the low-level implementation phase. Like DelTa, RD is defined at a level of abstraction that is independent from existing MTLs. Therefore, there are some similarities and differences between RD and DelTa. In RD, rules focus on mappings rather than constraints and actions in DelTa. The metamodel of 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. However, 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, which is covered by DelTa. The execution flow of RD supports sequencing rules, branching in alternative paths based on a constraint which is similar to the decision TUR in DelTa, or non-deterministically choosing to apply one rule which is similar to the choice TUR. They also allow rules to explicitly invoke the application of other rules. RD is inspired from QVT-R and ETL and is therefore more easily implemented in these languages.

Lano et al. [12] proposed TSPEC as the language to describe the structure of design patterns. The purpose of TSPEC is to formalize whole transformations, whereas the purpose of DelTa is to represent an abstraction of snippets of a transformation, i.e., transformation patterns. Whereas TSPEC uses mappings with constraints to represent rules in a transformation, DelTa provides a metamodel structure that lets create any kind of relation within the rules, including element mappings from source language to target language. TSPEC uses another metamodel, named language metamodel (LMM), to represent the languages on which the transformations operate upon. This is similar 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 to group rules together. DelTa provides both a graphical and textual concrete syntax, whereas TSPEC only provides a textual syntax. In conclusion, we can state that DelTa was designed intentionally from an engineering perspective, to help engineers to understand and implement patterns, and to generate transformations from it, whereas TSPEC formalizes the effects of a transformation and is used to analyze them.

5. Model Transformation Design Patterns

In this section, we apply the unified template to the identified model transformation design patterns. We only show three design patterns for the sake of readability and put the rest of them, including Lano's design patterns, in Appendix C. 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 here is to demonstrate applicability of the unified template and represent the solution of the design patterns in DelTa. Furthermore, we specify the category under which each pattern falls according to the classification of Lano et al.

5.1. 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 [20].

- **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 Fig. 3. The

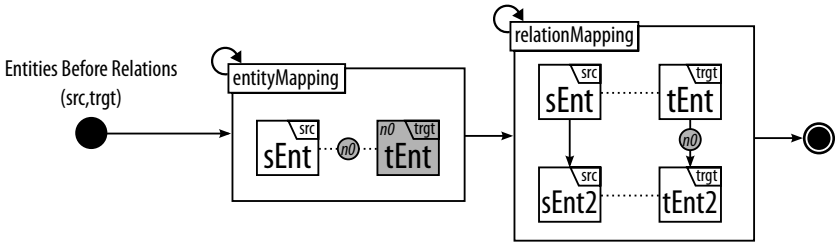

Figure 3: Entities before relations - Structure in DelTa

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, 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 [28].
- **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 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:**

```
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 attrs2cols {
    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 4: Rules of Entities before relations pattern in ATL

The implementation of the Entities before relations pattern in ATL

is depicted in Fig. 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 stipulate in this design pattern.

- **Related patterns:** The pattern can be identified as a special case of Phased Construction in Section Appendix C.3, 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. Fixed-point Iteration

This pattern falls under the "optimization" category.

- **Summary:** Fixed point iteration is a pattern for representing a "do-until" loop structure. 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 modification on the model, possibly at different locations: either adding new elements, removing elements, or modifying attributes.

- **Solution:** The solution is depicted in Fig. 5. 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 checkFixedPoint 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 are selected and the fixed point is checked again for a possible finding. If the rules in the block fail, it means no fixed-point is found and the result is a failure. The deleted elements are depicted in black in the Delete rule.

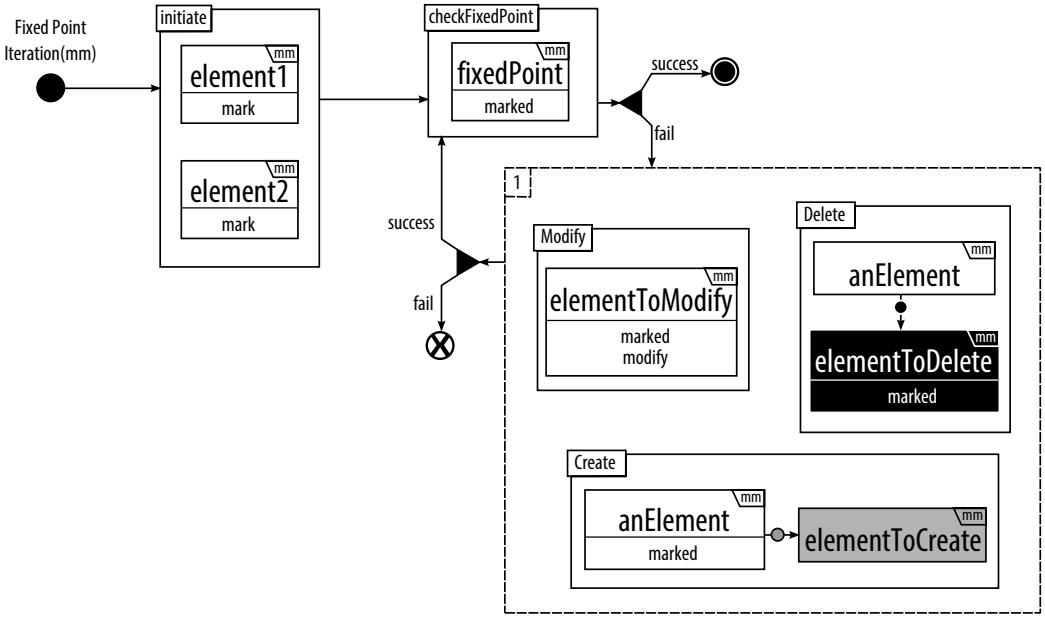

Figure 5: Fixed-point Iteration - Structure in DelTa

- **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 fields. For example in [34], 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.
- **Implementation:** Fig. 6 shows the implementation of the LCA from [34]

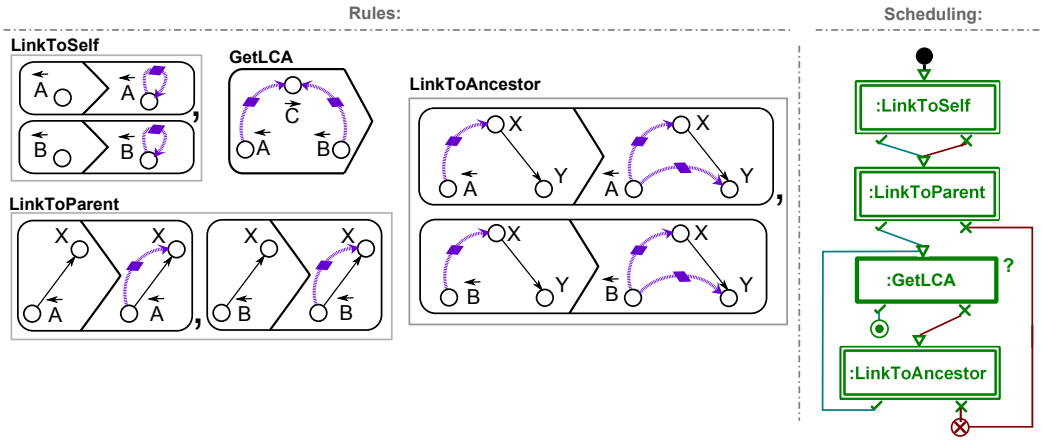

Figure 6: Rules and Scheduling in MoTif

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 rules implement 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. Also, auxiliary metamodel elements are 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.

5.3. Execution by Translation

This pattern falls under the "optimization" category.

- **Summary:** To execute a domain-specific language (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

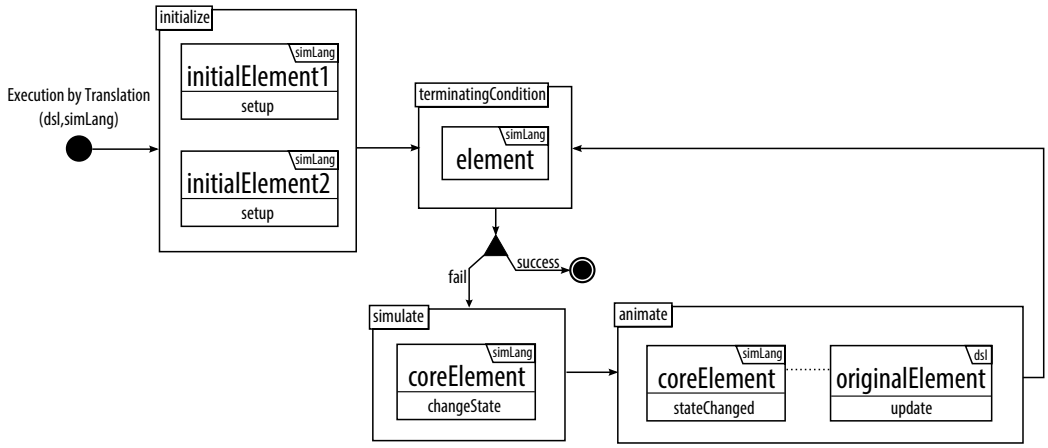

Figure 7: Execution by Translation - Structure in DelTa

the DSL we want to execute, and the `simLang`, which is the intermediate language we simulate instead of `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 goes on.

- **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 [37], Kühne et al. execute finite state automata (FSA) by translating to Petri Nets (PN). As they simulate the PN, they animate the FSA accordingly. In [43], we have defined a translation from activity diagram (AD) to 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 [44].

- **Implementation:** An implementation in MoTif is depicted in Fig. 8. The example maps PN to statecharts (SC) and uses the PN to simulate the SC. We only map the basic states and hyperedges in the SC for simplicity, but the advanced transformation can be found in [45]. 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 does the same job as in the previous examples. The `setOneTokenTo-Initial` rule puts one token to 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 to pass the matched elements between 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

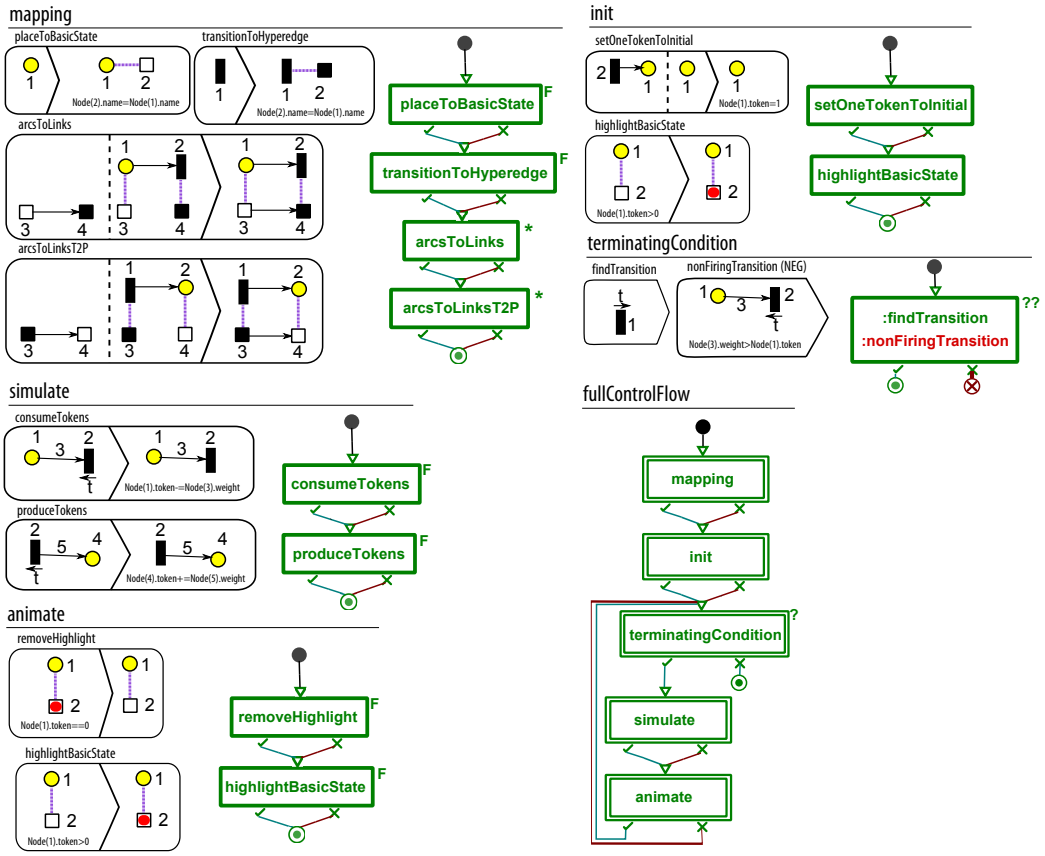

Figure 8: Petri Nets to Statecharts in MoTif

looks similar to the structure of the "execution by translation" design pattern. This is because we borrow the control flow of DelTa, which is TURelation, 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.

- **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.

6. Design Pattern-driven Model Transformation Development

The survey in Section 2 showed there is no systematic development process that model engineers follow when they want to solve a problem using model transformation, even if some preliminary processes have been proposed in the past [9, 46]. On the other hand, design patterns let model engineers reuse high-quality designs that have been proven to solve specific recurring problems by experienced practitioners. Therefore, there should be a way to let model engineers take advantage of these patterns when designing a model transformation. To remedy the ad-hoc design and implementation of model transformations (which consists of the large majority of projects in our pilot survey), we propose a design process to guide model engineers in their design choices by reusing design patterns in their implementations. The prototype we implemented showcases how this can be achieved by automatically instantiating patterns in the MTL of their choice using template-based code generation.

6.1. Process for designing and implementing model transformations

We describe the process model transformation engineers are encouraged to follow when they want to use design patterns in their transformation. Budinsky et al. [47] generated actual code from object-oriented design patterns and let the model engineers adapt the code to the rest of their application and further modify to add necessary application-specific details. We have adapted their approach to model transformation design patterns.

6.1.1. Problem identification

The very 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 complexity of the solution and thus the efficiency of the development [48]. A divide-and-conquer methodology has proven to be often useful to solve problems using model transformation because of the modularity of the rules and control structure this paradigm offers [49]. Larger problems can be decomposed into simpler sub-problems, until a solution using a design pattern becomes apparent.

6.1.2. Pattern selection

For each transformation (sub-)problem, the model engineer selects a model transformation design pattern that is best fit to solve it, as widely practiced

for object-oriented design pattern selection [50, 51]. This requires model engineers to scan through the design pattern catalog. However, senior and recurrent model engineers can only focus on the summary and application condition fields. 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.

6.1.3. Adaptation to 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: metamodels and elements involved, or adapting the semantics of tags (e.g., the example in Section 5.1 uses class diagrams as source metamodel). The adaptation step can also include investigating variants of the pattern, focusing on the variation fields (e.g., the mapping in Section 5.3 uses many-to-one instead of one-to-one mapping).

6.1.4. Implementation and refinement

At this point, the model engineer first implements as-is the customized pattern from the previous step. The choice of the MTL may require more effort at this step (e.g., if the MTL does not support explicit control scheduling, the model engineer also has to adopt the Simulating Explicit Rule Scheduling design pattern from Appendix C.3.9 in their transformations). Nevertheless, this step may be automated by generating a model transformation excerpt that implements the pattern [52]. Then, being 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 (e.g., the model engineer has to add more constraints to provide the logic for selecting the starting entity in the Visitor design pattern in Appendix C.1). Another refinement may be to add further rules to deal with different types (following the top-down phased construction in Appendix C.3.2) or to modularize the pattern (following the entity

splitting in Appendix C.3.5 and entity merging in Appendix C.3.6). More concrete examples can be found in [31].

6.1.5. Integration

The implemented pattern should be integrated carefully with the rest of the model transformation. Further customizations or modification may be required (e.g., the init and mapping phase of Petri Net to statecharts transformation in Fig. 8). In addition, micro-architectures can also be constructed by applying patterns in combination with each other [18].

6.1.6. 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 [53] or Agile Method [54]). This process does not assume there is necessarily a design pattern to solve the (sub-)problem at hand.

6.2. Benefits of a Design Pattern-driven Methodology

The core object-oriented design patterns have already demonstrated multiple benefits [55]. 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 [55]. 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 [56]. 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.

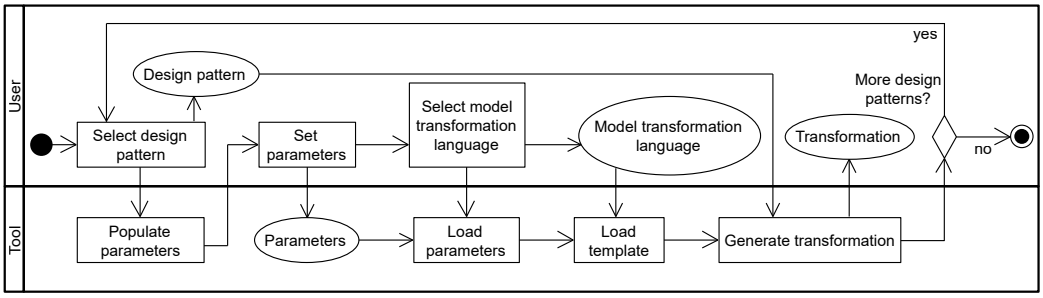

Figure 9: Activity diagram to generate a model transformation

6.3. Tool support to guide the model engineer

In order to alleviate the development process of model transformation for model engineers, we have implemented a tool that automates many cumbersome steps of the process described earlier when a design pattern is appropriate to solve the problem. The workflow of our transformation prototype is outlined by the activity diagram in Fig. 9 and a screenshot is illustrated in Fig. 10. The model engineer starts by selecting a design pattern; in this case, they selected the Entities before relations pattern. Each design pattern requires problem-specific parameters to be filled out by the model engineer (e.g., the rightmost area in Fig. 10 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.

We have implemented the prototype with a graphical user interface to simplify the process. The window depicted in Fig. 10 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 of the design pattern, following Budinsky et al.'s [47] code generation methodology. These parameters are inspired from transML rule diagrams [9] to bridge the gap between DelTa models and transformations in the specific

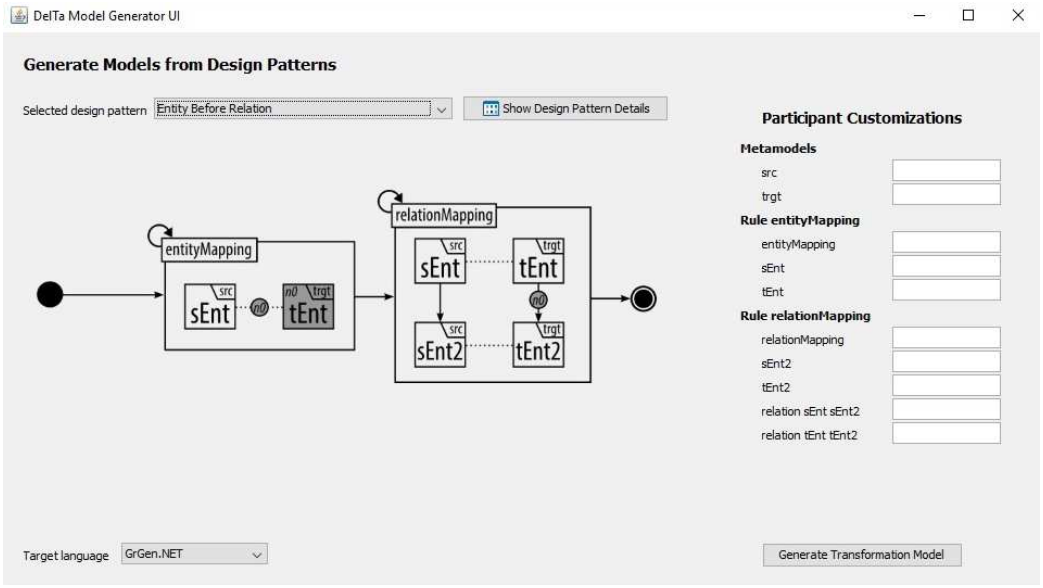

Figure 10: Design pattern generator tool

MTL: e.g., metamodel-specific type names. The editable parameter list is generated automatically from each DelTa model.

6.3.1. Implementation using MDE practices

Following MDE practices, we have adopted the Eclipse Modeling Framework (EMF) [57] 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 Fig. 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. We implemented the generation of model transformations using the template-based model-to-text transformation tool Xpand[5]. This is therefore a higher-order transformation [58]. In order to show the applicability of our approach, we support two different MTLs: GrGen.NET and Henshin. We choose these two languages because they are exogenous [4], which has different input and output metamodels. The choice of these two languages also demonstrates the breadth of applicability of DelTa in various MTLs.

[5]`https://eclipse.org/modeling/m2t/?project=Xpand`

6.3.2. Generation of concrete model transformations

We opted for a template-based code generation approach, as commonly practiced in MDE, to reduce the development effort for further design patterns and MTL support [4]. Only one template per MTL needs to be implemented, because a template depends only on the metamodel of DelTa. The template can be used to generate any design pattern from the catalog, as well as new ones expressed in DelTa. In our case, we implemented Xpand templates such that the interpreter takes as input a DelTa model in EMF and generates textual code that can be executed by a model transformation tool. Listing 1 depicts the general algorithm of the template to follow. It starts from the root node of the design pattern and traverses all elements hierarchically. Although this is a generic pseudo-template, small variations in the algorithm are necessary depending on the target transformation language. For example, when generating GrGen.NET transformation code, the template needs to create two files, one for the scheduling and another one for the rules, as in lines 2 and 19. however, only a single file is needed to generate the Henshin Ecore model. An additional step is required after each rule expansion (i.e., after line 17), where corresponding rule elements are mapped between the constraint and action to indicate that they refer to the same element.

Listing 1: General algorithm of a template to generation transformation code

```
1  Expand 'Model Transformation Design Pattern'
2    Create file for rules
3    Expand 'Pattern Metamodel'
4    Expand 'Transformation Unit'
5      Expand 'Rule'
6        Expand 'Annotation'
7        Expand 'Constraints'
8          Expand 'Variable'
9            Expand 'Annotation'
10           Expand 'Trace'
11           Expand 'Entity'
12           Expand 'Relation'
13           Expand 'Tag'
14       Expand 'Negative Constraints'
15         Expand 'Variable' under negative part of the rule
16       Expand 'Actions'
17         Expand 'Variable' under action part of the rule
18   Expand 'Start'
19     Create file for scheduling
```

```
20      Expand 'Sequence' for sequence after start
21        Expand 'Unit' for sequence after the next unit
22        Expand 'Decision'
23        Expand 'NoSched'
24        Expand 'Sequence' recursively for the next sequence
25        Expand 'End'
```

In the pseudo-code of Listing 1, the "`Expand`" keyword indicates to write the necessary textual code that corresponds to the DelTa element mentioned. For example, expanding a rule in line 5 means generating `rule` {...} for GrGen.NET and `<units xsi:type=''henshin:Rule''> ... </units>` for Henshin.

The implementation of the templates is straightforward, because DelTa is a domain-specific language whose elements often represent elements directly present in an MTL. Nevertheless, there were some challenges using this technology:

- Different parameters are needed for different transformation problems.We have solved this by creating extensions using XTend, which is another language commonly used together with Xpand. We could then parametrize our templates and set different parameters for each transformation case (e.g., the names of the design pattern, rules, variables while expanding) that the model engineer generates.

- Traversing the scheduling structure of the design pattern is not an intuitive task in Xpand, which is optimized to replace some textual equivalent version of each model element. However, in order to generate the scheduling part, we had to traverse the structure from the "start" node till the "end" node. Therefore, we have adopted a recursive approach. The scheduling part of the template starts whenever the `start` node is encountered. Then, we process each TUR structure as they are encountered until the `end` node is reached. However, when a `sequence` is found, we recursively expand the template responsible for TUR, as depicted on line 24.

- Assigning IDs to different elements of the model is another challenge we encountered. Even if we could assign an ID to an element randomly in Xpand, it is not possible to access it later on when we want to refer to that same element. Therefore, we generate an ID for an element so that it is possible to deterministically and uniquely assign and retrieve

its value based on the context of the element. For example, if an element of type `T` has to appear in the right-hand side of a rule `R` (i.e., post-condition of a graph transformation rule like in GrGen.NET and Henshin), its ID will be of the form `R_RHS_T`. Whenever we want to refer to that same element in the right-hand side, we can then easily recreate and reuse this ID.

An example output of the this tool is depicted in Fig. C.21. In that example, the transformation, that is using Visitor pattern, is automatically generated and then refined for the "class depth level" problem. The generated transformation model conforms to a specific MTL. We use the respective language compilers to establish this conformance check. For GrGen.NET, the verification consists of running the compiler with a command line. For Henshin, it consists of opening the model with the Henshin model editor that can only open valid models. We have successfully verified that all design patterns are correctly generated and conform to their respective MTLs.

6.4. Application on a Real Example

In what follows, we illustrate how model transformation design patterns are applied on the case study from the Transformation Tool Contest 2013 [59]: transforming a Petri net (PN) model to a Statechart (SC) model. We extend our original solution in MoTif [45] to execute the SC model using the underlying PN model. The solution is summarized in Section 5.3 under the implementation field. We solve this transformation problem following the process we proposed in Section 6.1.

6.4.1. Problem Identification

We choose to solve this case with a model transformation approach that uses an explicit scheduling of graph transformation rules, such as MoTif [7]. After analyzing the structure of each PN and SC, we propose to decompose the transformation into the following sub-problems:

1. **Initialization:** elements in each PN are mapped to elements in each SC.
2. **AND Reduction:** AND states are created in each SC for a special set of places in each PN.
3. **OR Reduction:** OR states are created in each SC for a special set of transitions in each PN.

4. **Control Flow:** overall control flow that depicts the ordering of the rules.
5. **Simulation:** execute each SC by simulating the corresponding PN.

6.4.2. Pattern Selection

We identify the most appropriate design pattern from the catalog for each transformation sub-problem. 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. In order to keep a tracing relation between SC and PN elements, we use the **Auxiliary Metamodel** pattern to create trace links. SC is a hierarchical structure, therefore, we use the **Top-down 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 has a suitable scenario 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. Thus, we use the **Object Indexing** pattern. The overall simulation of the SC using the corresponding PN is ensured using the **Execution by Translation** pattern.

6.4.3. Adaptation to Problem

For most of the previously identified design patterns, the source metamodel is a PN and the target metamodel is a SC. In addition, all the names in the patterns should be adapted to represent the appropriate PN and SC structures. For the initialization phase, we need a one-to-many variant of the Entities before relations pattern, because an element in a PN is mapped to two elements in a SC. No variation is needed for the remaining design patterns.

6.4.4. Implementation and Refinement

We have implemented the problem in MoTif. Below, we highlight how each of the previously adapted design patterns are implemented in the solution. The complete final transformation is available in the report [45].

Entities Before Relations:. These rules map the elements of a PN to elements in a SC while maintaining traceability links between the source and the target. In Fig. 11, the elements of a PN that are "place," "transition," and

"arcs" are mapped to the elements of a SC that are "BasicOR," "hyperedge," and "links."

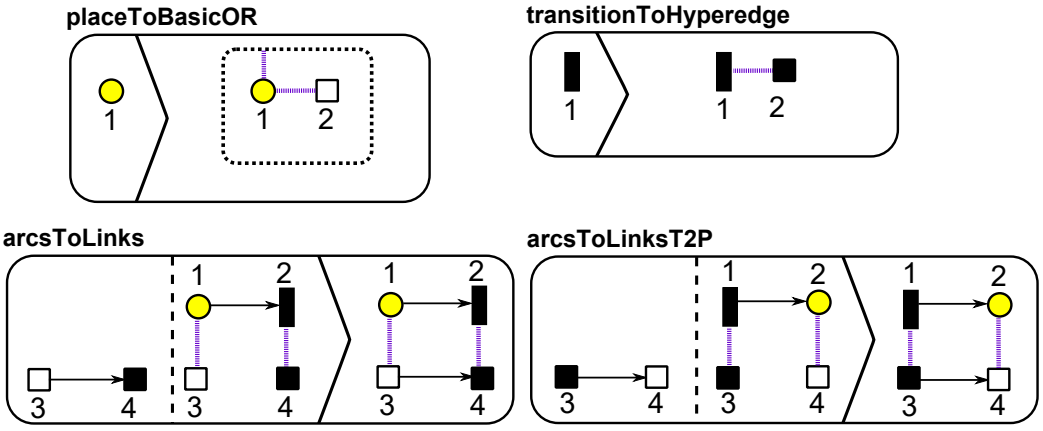

Figure 11: Mapping Phase Rules

Parallel Composition:. The mapping rules in Fig. 11 can be executed in parallel because they do not share any common elements; e.g., rules "placeToBasicOR" and "transitionToHyperEdge" can be performed in parallel.

Auxiliary Metamodel:. The dashed links in Fig. 11 represents the trace links that are part of a separate metamodel, i.e., auxiliary metamodel. As a functionality of the MoTif language, trace links belong to neither the source nor the target metamodels.

Top-down Phased Construction:. The OR states are created in the "placeToBasicOR" rule of Fig. 11. Following the hierarchy, the new elements are sub-states of the OR state in the rule in Fig. 12.

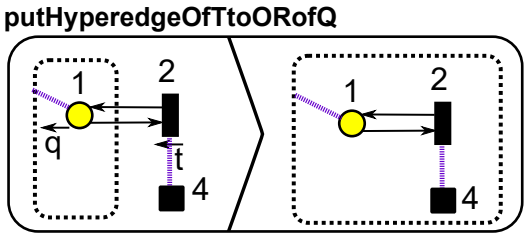

Figure 12: A Rule from OR Reduction Phase

Object Indexing:. In MoTif, we use pivots to index elements and refer to them later on. One example is pivot $\vec{t}$ for the transition labeled as 2 in Fig. 12. This pivot was set in a previous rule and is bound to it by reference in this rule.

Fixed-point Iteration:. The control flow in Fig. 13 applies the Fixed-point Iteration pattern twice. For both AND and OR reductions in the transformation, we continue to reduce the SC as long as there are more reduction points. In this case, the fixed point condition to check is whether there are no further possible reductions.

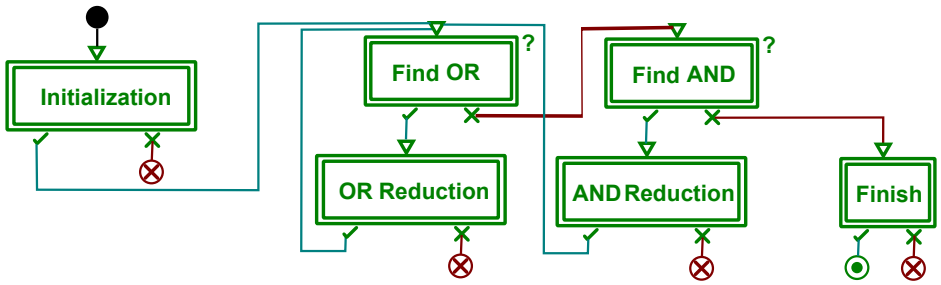

Figure 13: Overall Control Flow

Execution by Translation:. The overall transformation is an instance of the Execution by Translation pattern. The more abstract the design pattern, the more it requires manual refinement by the model engineer. The Execution by Translation pattern is an example of this situation. The tool generates the partial transformation of this pattern, mainly focusing on the control flow of the rules containing simple patterns. Fig. 14 depicts the rules and control flow of the simulation after the manual refinement. The terminating condition of the pattern is the PN having a firing transition. The simulate part of the pattern consumes and produces tokens. Finally, the animate part highlights the elements that contain tokens.

7. Validation User Study

We conducted a user study to understand the validity of the design pattern driven methodology we have introduced in Section 6. The research questions identified are:

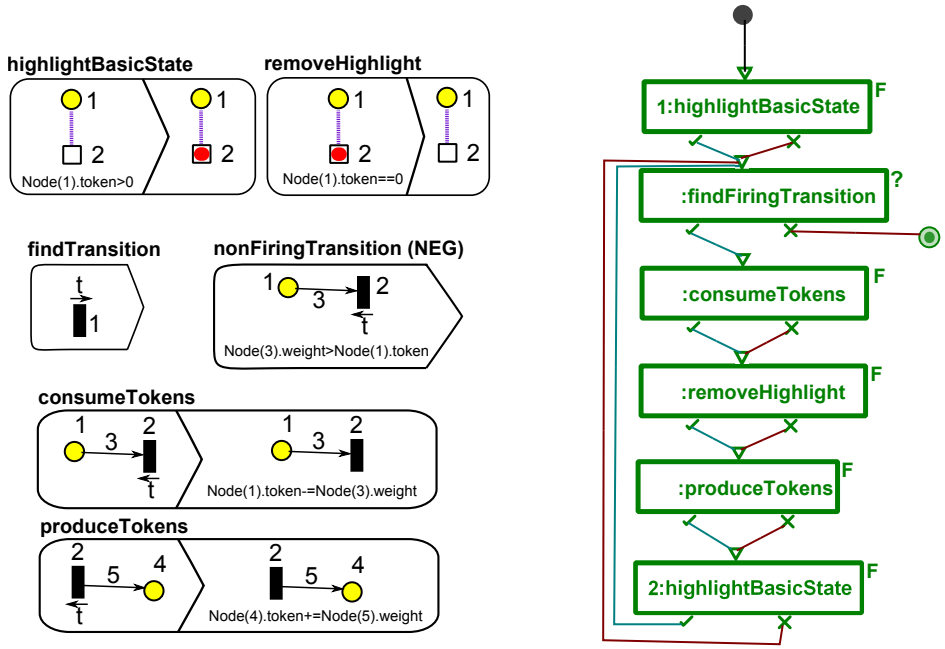

Figure 14: Simulation

RQ1 Does the methodology have any impact on the model engineers who are trying to solve a model transformation problem?

RQ2 Is the tool prototype given in Section 6.3 useful for model engineers?

7.1. Experimental Setup

The study consisted of solving a model transformation problem using a specific MTL: GrGen.Net [5]. We prepared a remote Windows machine that had all the necessary software installed on Amazon EC2[6]. Participants only had to connect to the machine and follow online directives we provided to them in order to take the study. The directives were divided into the following parts:

1. Follow a quick GrGen.NET tutorial teaching the language basics covering the necessary structures needed to solve the problem.
2. Read the description of the problem they needed to solve and start thinking of a solution.

[6] `http://aws.amazon.com/ec2/`

3. Read the description of the design pattern driven methodology (see Section 6.1).
4. Follow a tutorial on how to use the Design Pattern Generator tool (see Section 6.3).
5. Solve the problem using the tool.
6. Complete a survey regarding their experience in the user study.

Participants had a two-hour time slots alloted to them with access to resources we provided them at any time. The problem to solve was a simple translation from C language structures connected with pointers to Java language classes connected with inheritance links and associations. The problem had two parts. The first part was the translation so that the resulting Java model had classes with single inheritance only. The second part was the computation of depth levels of each class in the inheritance hierarchy. The depth level represents the number of classes between a class and its furthest ancestor. We provided them with an initial project with simplified metamodels of C and Java, along with the tools necessary for training purposes. The post-survey consisted of 5 questions. We asked a question about their experience with GrGen.NET. Then, we asked them whether the methodology had any impact on their conceptual thinking for a solution to the problem. The remaining questions asked the participants to rate various properties of the methodology, the tool, and the DelTa language on a scale of 1 to 5 (as "bad", "poor", "average", "good", and "excellent").

7.2. Data Collection

We collected and analyzed the actual transformation solutions after each participant complete his study. The post-survey is available online[7]. We again used the Qualtrics software to analyze the results of the survey.

7.3. 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 developers 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] `http://tinyurl.com/UserStudy2016`

7.4. Results of the user study

Had impact	Result
Yes	7
No	3

Table 6: Effect of the methodology

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

Table 7: 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 8: Ratings of the properties

7.4.1. RQ1: Impact of the methodology on model engineers

The setup was such that participants were first given the problem first, and only then was the methodology and design patterns revealed with minimal training. When solving the problem, they had to choose the most appropriate design patterns from the ones available in the tool. This setup was to reduced the probability of bias with the methodology when asked about it.

Table 6 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 they claimed it did not influence them.

Table 8 summarizes how they 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 of them 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 reports how many participants completed successfully each task. 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.

7.4.2. RQ2: Usefulness of the design pattern generator tool

The tool generates a partial transformation from a selected design pattern. Besides 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 correctly solve the problem.

Table 8 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 2. 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 by itself is only one part of the

design pattern definition, but they relied only on that. 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.

7.5. 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 have 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 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 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.

8. Conclusion

We conclude the paper with a summary and a discussion on the limitation of our approach. Finally, we discuss future outlooks on adopting model transformation design patterns as well as ideas to extend the work we presented.

8.1. Summary

We surveyed model transformation engineers to understand the needs for model transformation design patterns and the essential requirements for a language to express them. After analyzing existing model transformation

design pattern studies, we noted no consensus on the meaning of a model transformation design pattern and how to represent it. Therefore, 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, and it can be used to define all 14 existing design patterns as well as new 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. We have also implemented a tool that allows generation of model transformation excerpts from patterns expressed in DelTa in a concrete MTL to help and guide model engineers in their design and implementation. Finally, we have validated the tool along with the DelTa language and the methodology with a user study.

8.2. Limitations

Instead of DelTa, a formal specification language such as in [60] 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 [61]), rule-based (MoTif [7]), relational (QVT-R [62]), using term rewriting (Stratego [63]), template-based (Xpand [64]), or by-example [65]. We only focus on rule-based transformations.

8.3. 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 Section 5 and Appendix C 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 Section 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 [66]. 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.

8.4. Future Work

Our implementation proved 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 and ATL, and bi-directional transformations, such as QVT-Relations and Triple Graph Grammars. 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 [67], we need to further investigate design patterns applicable these kinds of problems. Although the initial study in Section 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 are already trained with design patterns for model transformations. Therefore, it would be ideal to integrate the findings of this paper in advanced MDE courses. Finally, as pointed out in Section 8.3, 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.

References

[1] T. Stahl, M. Voelter, K. Czarnecki, Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2006.

[2] D. Harel, B. Rumpe, Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff, Tech. rep., Weizmann Institute Of Science (2000).

[3] L. Lucio, M. Amrani, J. Dingel, L. Lambers, R. Salay, G. M. Selim, E. Syriani, M. Wimmer, Model Transformation Intents and Their Properties, Journal on Software and Systems Modeling (2014) 1–38.

[4] K. Czarnecki, S. Helsen, Feature-Based Survey of Model Transformation Approaches, IBM Systems Journal 45 (3) (2006) 621–645.

[5] R. Geiß, M. Kroll, GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool, in: Applications of Graph Transformations with Industrial Relevance, Vol. 5088 of LNCS, Kassel, Germany, 2008, pp. 568–569.

[6] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations, in: Model Driven Engineering Languages and Systems, Vol. 6394 of LNCS, Oslo, Norway, 2010, pp. 121–135.

[7] E. Syriani, H. Vangheluwe, A Modular Timed Model Transformation Language, Journal on Software and Systems Modeling 12 (2) (2011) 387–414.

[8] J. Hutchinson, J. Whittle, M. Rouncefield, S. Kristoffersen, Empirical Assessment of MDE in Industry, in: International Conference on Software engineering, Honolulu, HI, 2011, pp. 471–480.

[9] E. Guerra, J. de Lara, D. Kolovos, R. Paige, O. dos Santos, Engineering Model Transformations with transML, Journal on Software and Systems Modeling 12 (3) (2011) 555–577.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley Professional, 1994.

[11] A. Agrawal, Reusable Idioms and Patterns in Graph Transformation Languages, in: International Workshop on Graph-Based Tools, Vol. 127 of Electronic Notes in Theoretical Computer Science, 2005, pp. 181–192.

[12] K. Lano, S. Kolahdouz Rahimi, Model-Transformation Design Patterns, IEEE Transactions on Software Engineering 40 (12) (2014) 1224–1259.

[13] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, Software Patterns Series, Pearson Education, 2004.

[14] M. Dwyer, G. Avrunin, J. Corbett, Patterns in Property Specifications for Finite-state Verification, in: Proceedings of the International Conference on Software Engineering, Los Angeles, CA, USA, 1999, pp. 411–420.

[15] D. Spinellis, Notable Design Patterns for Domain-specific Languages, Journal of Systems and Software 56 (1) (2001) 91 – 99.

[16] H. Cho, J. Gray, Design Patterns for Metamodels, in: SPLASH '11 DSM Workshop, Portland, OR, USA, 2011, pp. 25–32.

[17] W. Brown, R. Malveau, S. McCormick, T. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, 1998.

[18] K. Beck, R. Johnson, Patterns Generate Architectures, in: Object-Oriented Programming, Vol. 821 of LNCS, Bologna, Italy, 1994, pp. 139–149.

[19] S. M. Yacoub, H. H. Ammar, Pattern-oriented Analysis and Design: Composing Patterns to Design Software Systems, Addison-Wesley Professional, 2004.

[20] M.-E. Iacob, M. W. A. Steen, L. Heerink, Reusable Model Transformation Patterns, in: Enterprise Distributed Object Computing Workshop, IEEE Computer Society, Munich, Germany, 2008, pp. 1–10.

[21] J. Bézivin, F. Jouault, J. Paliès, Towards Model Transformation Design Patterns, in: Proceedings of the First European Workshop on Model Transformations, 2005.

[22] A. Le Guennec, G. Sunye, J.-M. Jezequel, Precise Modeling of Design Patterns, in: UML 2000 - The Unified Modeling Language, Vol. 1939 of LNCS, York, UK, 2000, pp. 482–496.

[23] E. Syriani, J. Gray, Challenges for Addressing Quality Factors in Model Transformation, in: Software Testing, Verification and Validation, ICST'12, IEEE, 2012, pp. 929–937.

[24] Y. Aridor, D. B. Lange, Agent Design Patterns: Elements of Agent Application Design, in: Proceedings of the Second International Conference on Autonomous Agents, AGENTS '98, ACM, 1998, pp. 108–115.

[25] A. Gangemi, V. Presutti, Ontology Design Patterns, in: Handbook on Ontologies, International Handbooks on Information Systems, Springer, 2009, pp. 221–243.

[26] A. DeHon, J. Adams, M. deLorimier, N. Kapre, Y. Matsuda, H. Naeimi, M. Vanier, M. Wrighton, Design Patterns for Reconfigurable Computing, in: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM, 2004, pp. 13–23.

[27] H. Ergin, E. Syriani, Towards a Language for Graph-Based Model Transformation Design Patterns, in: Theory and Practice of Model Transformations, Vol. 8568 of LNCS, Springer, 2014, pp. 91–105.

[28] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, ATL: A Model Transformation Tool, Science of Computer Programming 72 (1-2) (2008) 31–39.

[29] D. Kolovos, R. Paige, F. Polack, The Epsilon Transformation Language, in: International Conference on Model Transformations, Vol. 5063 of LNCS, Zürich, Switzerland, 2008, pp. 46–60.

[30] I. Kurtev, State of the Art of QVT: A Model Transformation Language Standard, in: Applications of Graph Transformations with Industrial Relevance, Vol. 5088 of Lecture Notes in Computer Science, Kassel, Germany, 2008, pp. 377–393.

[31] H. Ergin, E. Syriani, Implementations of Model Transformation Design Patterns Expressed in DelTa, Tech report SERG-2014-01, University of Alabama, Department of Computer Science (Feb 2014).

[32] T. Mens, P. Van Gorp, A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science 152 (2006) 125 – 142.

[33] T. Levendovszky, L. Lengyel, T. Mszros, Supporting Domain-specific Model Patterns with Metamodeling, Journal on Software and Systems Modeling 8 (4) (2009) 501–520.

[34] H. Ergin, E. Syriani, Identification and Application of a Model Transformation Design Pattern, in: ACM Southeast Conference, ACMSE'13, Savannah, GA, 2013.

[35] E. Syriani, J. Gray, H. Vangheluwe, Modeling a Model Transformation Language, in: Domain Engineering: Product Lines, Conceptual Models, and Languages, 2012, pp. 211–237.

[36] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science, 2006.

[37] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, M. Wimmer, Explicit Transformation Modeling, in: MODELS 2009 - Models in Software Engineering Workshop, Vol. 6002 of LNCS, Denver, CO, USA, 2010, pp. 240–255.

[38] G. Taentzer, AGG: A Graph Transformation Environment for Modeling and Validation of Software, in: Applications of Graph Transformations with Industrial Relevance, Vol. 3062 of Lecture Notes in Computer Science, Charlottesville, VA, USA, 2004, pp. 446–453.

[39] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf, Model Transformation with a Visual Control Flow Language, International Journal of Computer Science 1 (1) (2006) 45–53.

[40] E. Syriani, H. Vangheluwe, De-/Re-constructing Model Transformation Languages, European Association of Software Science and Technology (2010) 29.

[41] A. Agrawal, G. Karsai, F. Shi, Graph Transformations on Domain-specific Models, Journal on Software and Systems Modeling (2003).

[42] D. Varro, A. Balogh, The Model Transformation Language of the {VIATRA2} Framework, Science of Computer Programming 68 (3) (2007) 214 – 234.

[43] E. Syriani, H. Ergin, Operational Semantics of UML Activity Diagram: An Application in Project Management, in: Requirements Engineering Conference 2012 Workshops: Model-driven Requirements Engineering, Chicago, IL, USA, 2012, pp. 1–8.

[44] J. de Lara, H. Vangheluwe, Automating the Transformation-based Analysis of Visual Languages, Formal Aspects of Computing 22 (3-4) (2010) 297–326.

[45] H. Ergin, E. Syriani, AToMPM Solution for the Petri Net to Statecharts Case Study, in: Seventh Transformation Tool Contest, 2013.
URL `http://hergin.students.cs.ua.edu/research/ttc2013.pdf`

[46] A. Balogh, G. Bergmann, G. Csertan, L. Gonczy, A. Horvath, I. Majzik, A. Pataricza, B. Polgar, I. Rath, D. Varro, G. Varro, Workflow-Driven Tool Integration Using Model Transformations, in: Graph Transformations and Model-Driven Engineering, Vol. 5765 of Lecture Notes in Computer Science, 2010, pp. 224–248.

[47] F. Budinsky, M. Finnie, J. Vlissides, P. Yu, Automatic Code Generation from Design Patterns, IBM Systems Journal 35 (2) (1996) 151–171.

[48] A. Rensink, P. Van Gorp, Graph Transformation Tool Contest 2008, International Journal on Software Tools for Technology Transfer 12 (3-4) (2010) 171–181.

[49] E. Syriani, H. Vangheluwe, A Modular Timed Graph Transformation Language for Simulation-based Design, Journal on Software and Systems Modeling 12 (2) (2013) 387–414.

[50] B. Zamani, G. Butler, S. Kayhani, Tool Support for Pattern Selection and Use, Electronic Notes in Theoretical Computer Science 233 (2009) 127–142.

[51] S. M. H. Hasheminejad, S. Jalili, Design Patterns Selection: An Automatic Two-phase Method, Journal on Software and Systems Modeling 85 (2) (2012) 408–424.

[52] H. Albin Amiot, P. Cointe, Y.-G. Guhneuc, N. Jussien, Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together, in: Automated Software Engineering, ASE'01, Coronado Island, San Diego, CA, 2001, pp. 166–173.

[53] G. Booch, I. Jacobson, J. Rumbaugh, The Unified Software Development Process, Addison Wesley, 1999.

[54] R. C. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall PTR, 2003.

[55] E. Agerbo, A. Cornils, How to preserve the benefits of design patterns, SIGPLAN Not. 33 (10) (1998) 134–143.

[56] F. P. Brooks, Jr., No Silver Bullet Essence and Accidents of Software Engineering, Computer 20 (4) (1987) 10–19.

[57] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: Eclipse Modeling Framework, Pearson Education, 2008.

[58] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, J. Bézivin, On the Use of Higher-Order Model Transformations, in: European Conference on Model Driven Architecture: Foundations and Applications, Vol. 5562 of Lecture Notes in Computer Science, Enschede, The Netherlands, 2009, pp. 18–33.

[59] P. Van Gorp, L. M. Rose, The Petri-Nets to Statecharts Transformation Case, in: Sixth Transformation Tool Contest, Vol. 135, Budapest, Hungary, 2013, pp. 16–31.

[60] K. Lano, S. Kolahdouz Rahimi, Constraint-based Specification of Model Transformations, Journal of Systems and Software 86 (2) (2013) 412–436.

[61] J.-R. Falleri, M. Huchard, C. Nebut, Towards a Traceability Framework for Model Transformations in Kermeta, in: ECMDA-TW'06: European Conference on Model-Driven Architecture Traceability Workshop, 2006, pp. 31–40.

[62] Object Management Group, Meta Object Facility 2.0 Query/View/-Transformation Specification (Jan 2011).

[63] E. Visser, Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5, in: Rewriting Techniques and Applications, Vol. 2051 of Lecture Notes in Computer Science, Utrecht, The Netherlands, 2001, pp. 357–361.

[64] B. Klatt, Xpand: A Closer Look at the Model2text Transformation Language, Language 10 (16).

[65] D. Varro, Model Transformation by Example, in: Model Driven Engineering Languages and Systems, Vol. 4199 of Lecture Notes in Computer Science, Genova, Italy, 2006, pp. 410–424.

[66] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. Halkidis, Design Pattern Detection Using Similarity Scoring, Transactions on Software Engineering 32 (11) (2006) 896–909.

[67] Edouard Batot, Houari Sahraoui, Eugene Syriani, Paul Molins, Wael Sboui, Systematic Mapping Study of Model Transformations for Concrete Problems, in: Model-Driven Engineering and Software Development, SciTePress, 2016, pp. 176–183.

Appendix A. DelTa Textual Concrete Syntax

We have designed an alternative textual concrete syntax for DelTa. Listing 2 shows the EBNF grammar implemented in Xtext.

Listing 2: EBNF Grammar of DelTa in XText

```
MTDP:
    'mtdp' NAME
        'metamodels:' NAME (',' NAME) *  ANNOTATION?
        ('rule' NAME '*' ?  ANNOTATION?
                Entity?
                Relation?
                Trace?
                Constraint
                NegativeConstraint*
                ForbiddenConstraint*
                Action) +
        TURelation+ ;

```

```
Entity: 'Entity' ELEMENTNAME (',' ELEMENTNAME)* ;
Relation: 'Relation' NAME '(' ELEMENTNAME ',' ELEMENTNAME ')'
              (',' NAME '(' ELEMENTNAME ',' ELEMENTNAME ')') * ;
Trace: 'Trace' NAME '(' ELEMENTNAME (',' ELEMENTNAME) + ' ) '
              (',' NAME '(' ELEMENTNAME (',' ELEMENTNAME) + ' ) ' ) * ;
Constraint: 'constraint:' '~'? (ELEMENTNAME|NAME ) ( ',' '~'? ( ELEMENTNAME|NAME ) ) *
                    ANNOTATION? ;
NegativeConstraint: 'negative constraint:'(ELEMENTNAME|NAME) ( ',' ( ELEMENTNAME|NAME ) ) *
                    ANNOTATION? ;
ForbiddenConstraint: 'forbidden constraint:'(ELEMENTNAME|NAME) ( ',' ( ELEMENTNAME|NAME ) ) *
                    ANNOTATION? ;
Action: ( 'action:' ('~' ? ( ELEMENTNAME|NAME ) ( ','~' ? ( ELEMENTNAME|NAME ) ) * ) )
              ANNOTATION? ;
TURelation: (TURTYPE ( 'START' | ( NAME ( ' [ 'NAME'=' ( 'true' | 'false' ) ' ] ' ) ? ) )
              (',' ( 'END' | NAME ) ( ' [ 'NAME'=' ( 'true' | 'false' ) ' ] ' ) ? ) + )
              | Decision;
Decision: NAME '?' DecisionBlock ':' DecisionBlock;
DecisionBlock: ('END' | NAME ) ( ' [ ' ( 'END' | NAME ) '=' ( 'true' | 'false' ) ' ] ' ) ?
        (',' ( 'END' | NAME ) ( ' [ ' ( 'END' | NAME ) '=' ( 'true' | 'false' ) ' ] ' ) ? ) * ;
terminal NAME: ('a'..'z' | 'A'..'Z' ) ( 'a'..'z' | 'A'..'Z' | '0'..'9' ) * ;
terminal ELEMENTNAME: NAME '.' NAME ('[' NAME '=' ('true' | 'false' )
              (',' NAME '=' ('true' | 'false' ) ) * ' ] ' ) ? ;
terminal ANNOTATION: '#' (!'#' ) * '#' ;
terminal TURTYPE: ('Sequence' | 'Choice' | 'Parallel' | 'NoSched' ) ':' ;
```

Listing 3 shows the Entities before relations pattern explained in Section 5 in the textual concrete syntax.

Listing 3: Entities before relations MTDP

```
mtdp EntitiesBeforeRelations
    metamodels: src, trgt
    rule entityMapping*
        Entity src.e, trgt.f
        Trace t1(src.e, trgt.f)
        constraint: src.e, ~trgt.f, ~t1
        action: trgt.f, t1
    rule relationMapping*
        Entity src.e, src.f, trgt.g, trgt.h
        Relation r1(src.e, src.f), r2(trgt.g, trgt.h)
        Trace t1(src.e, trgt.g), t2(src.f, trgt.h)
        constraint: src.e, src.f, trgt.g, trgt.h, r1, t1, t2, ~r2
        action: r2
    Sequence: START, entityMapping, relationMapping, END
```

Appendix B. DelTa Graphical Concrete Syntax

In this section, we present all the graphical concrete syntax of DelTa exhaustively.

Appendix C. Model Transformation Design Patterns - Cont'd

In this section, we continue listing the design patterns in unified template.

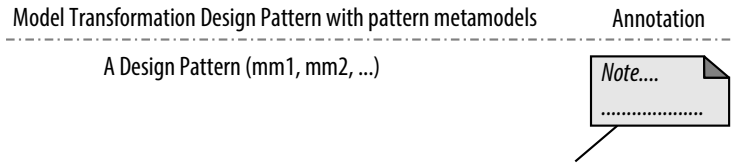

Figure B.15: Model Transformation Design Pattern and Annotation

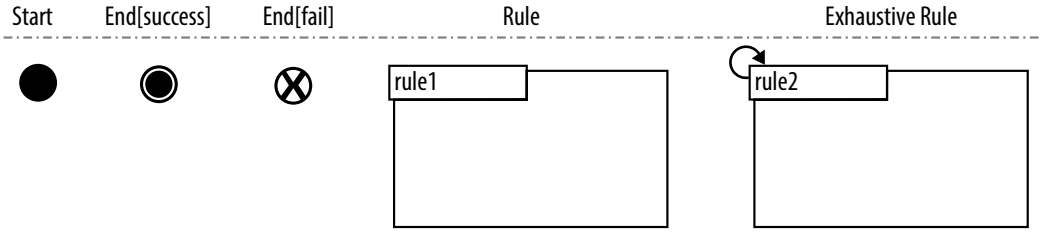

Figure B.16: Transformation Units

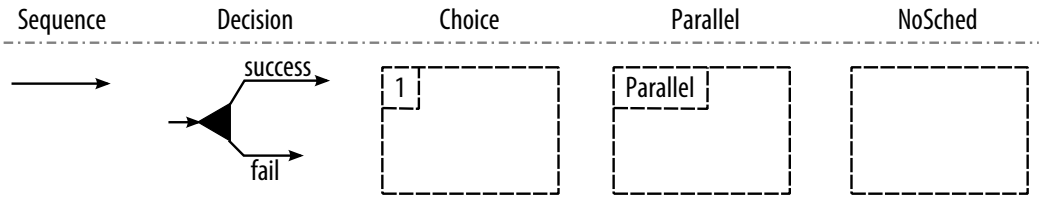

Figure B.17: Transformation Unit Relations

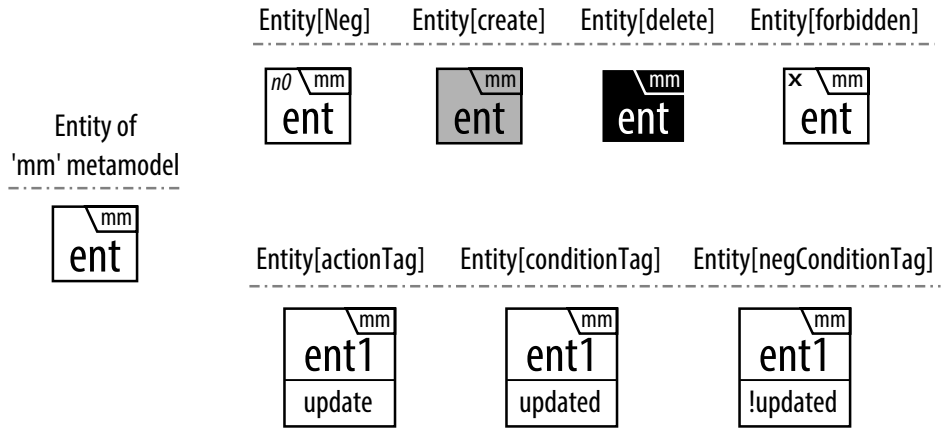

Figure B.18: Pattern Metamodel - Entity

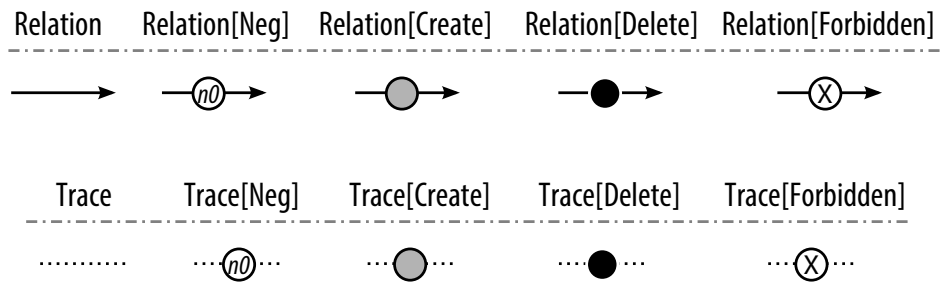

Figure B.19: Pattern Metamodel - Relation and Trace

Appendix C.1. Visitor

This pattern falls under the "rule modularization" category.

- **Summary:** This pattern traverses all the nodes in a tree and processes each entity individually [27].
- **Application Conditions:** The 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 pattern is depicted in Fig. C.20. The 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.
- **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.

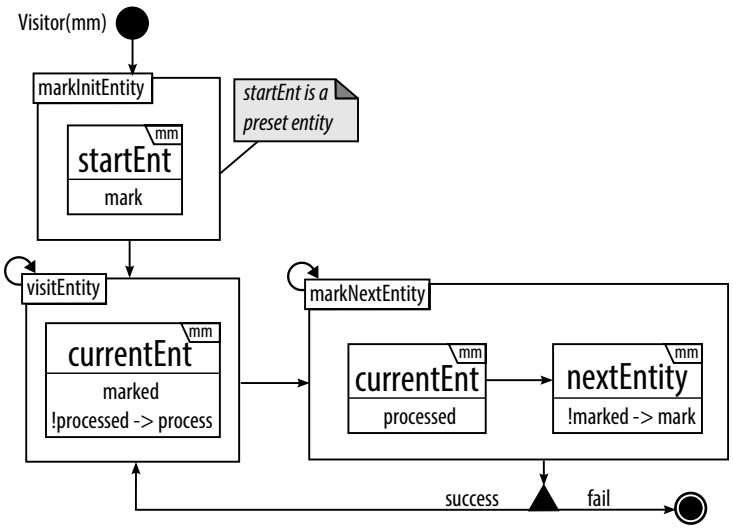

Figure C.20: Visitor - Structure in DelTa

- **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:** Fig. C.21 depicts an implementation of the visitor

```
rule markBaseClass {
    e:Class;
    negative {
        d:Class;
        d-:subclass->e;
    }
    modify {
        eval {
            e.marked=true;
            e.processed=true;
        }
    }
}
```

```
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;
        }
    }
}
```

```
rule markSubclass {
    e:Class;
    f:Class;
    e-:subclass->f;
    if {
        e.processed==true;
        f.marked==false;
    }
    modify {
        eval {
            f.marked=true;
        }
    }
}
```

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

Figure C.21: Visitor rules and scheduling in GrGen.NET

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 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 Fig. C.21. 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 [12], 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.

Appendix C.2. Transitive Closure

This pattern falls under the "rule modularization" category.

- **Summary:** Transitive closure is a pattern typically used for analyzing reachability related problems with an in-place transformation. It was proposed as a pattern in [11] and in [33]. 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 Fig. C.22. 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 connected with a trace. It is also applied recursively to cover all nodes exhaustively.

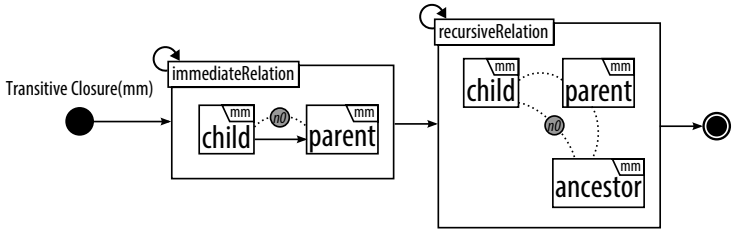

Figure C.22: Transitive Closure - Structure in DelTa

- **Benefits:** Since all the trace elements are created from each element to all its ancestors, queries relying on this information lookups 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 which 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 UML class diagram.

- **Implementation:** We have implemented the transitive closure in AGG. Fig. C.23 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

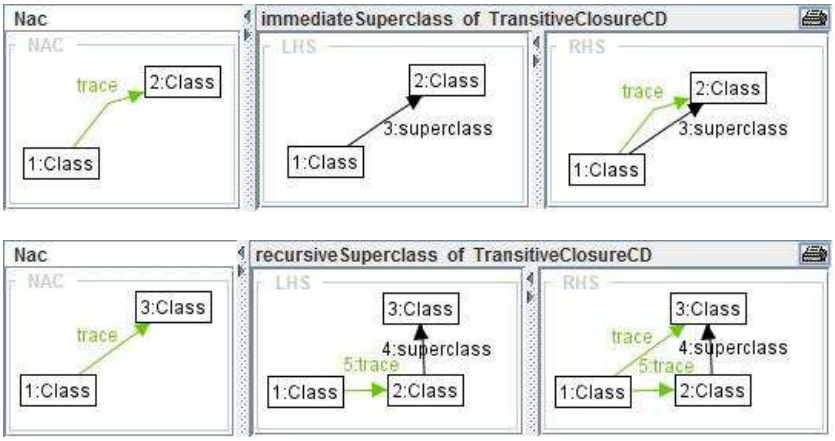

Figure C.23: Transitive Closure rules in AGG

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 executing 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 to be applied in a sequence.

- **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 metamodel if allowed. Different types of relations can be used to provide a priority structure.

Appendix C.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. [12] in DelTa concrete syntax. We only present the summary and solution fields from the unified template, because the complete description of the design pattern is already provided in the original paper.

Appendix C.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." [12]
- **Solution:** The solution is depicted in Fig. C.24. In the "firstRule", an entity is marked by setting a flag and in the "secondRule," the same entity is used.

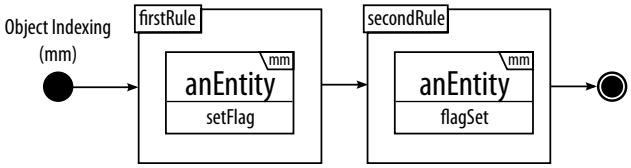

Figure C.24: Object Indexing - Structure in DelTa

- **Variation:** Some MTLs provide internal mechanisms to support this design pattern (e.g., pivot structure in MoTif [7], GReAT [41] and VMTS [39]).

Appendix C.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." [12]
- **Solution:** The solution is depicted in Fig. C.25. In the "formerPhase" rule, a container element "tContainer" of target metamodel is created and in the "latterPhase," its composite element "tComposite" is created.

Appendix C.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.

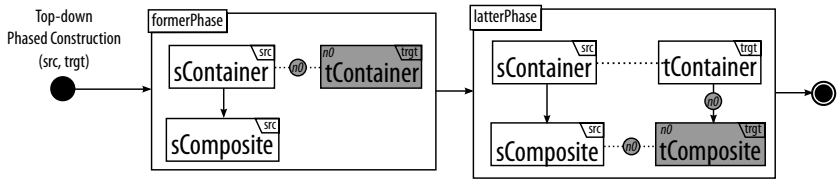

Figure C.25: Top-down Phased Construction - Structure in DelTa

- **Solution:** The solution is depicted in Fig. C.26. 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.

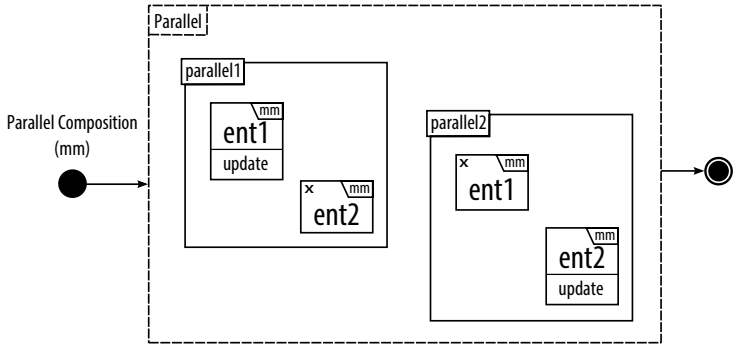

Figure C.26: Parallel Composition - Structure in DelTa

Appendix C.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 Fig. C.27. If "someEnt" element is created in a rule to be chosen from a group of rules, which are put inside a "NoSched" TUR, then it should not be created in another rule, which violates "someEnt"s being unique.

Appendix C.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.

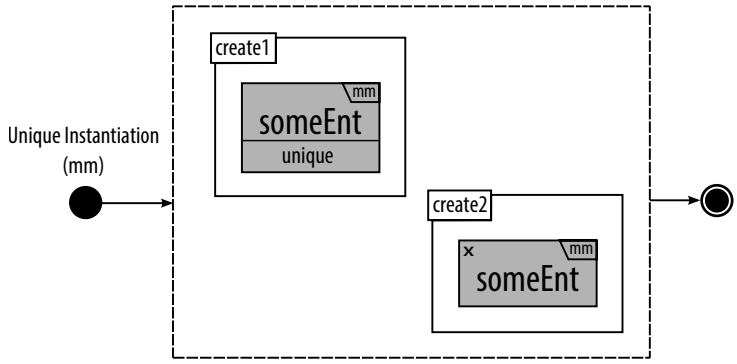

Figure C.27: Unique Instantiation - Structure in DelTa

- **Solution:** The solution is depicted in Fig. C.28. 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.

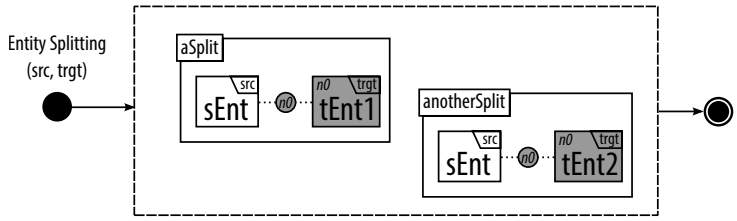

Figure C.28: Entity Splitting - Structure in DelTa

Appendix C.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 occur within its separate rule.

- **Solution:** The solution is depicted in Fig. C.29. 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.

Appendix C.3.7. Construction & Cleanup

- **Summary:** "This pattern structures a transformation by separating rules which construct model elements from those which delete ele-

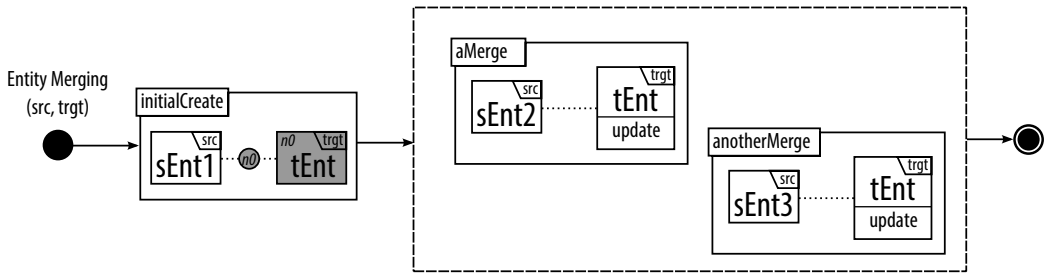

Figure C.29: Entity Merging - Structure in DelTa

ments." [12]

- **Solution:** The solution is depicted in Fig. C.30. 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.

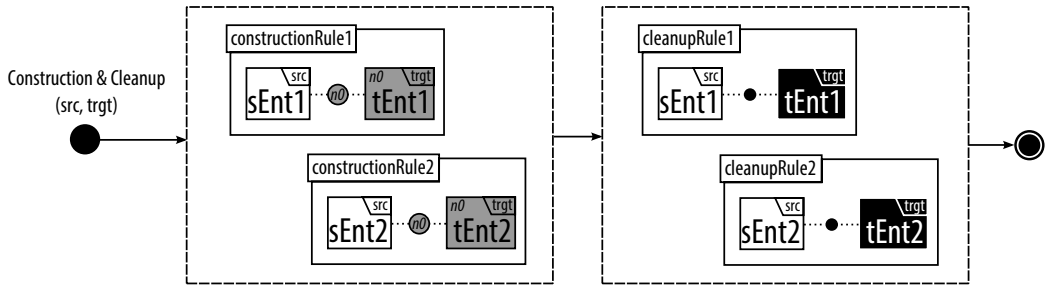

Figure C.30: Construction & Cleanup - Structure in DelTa

Appendix C.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 metamodels.

- **Solution:** The solution is depicted in Fig. C.31. If any of 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.

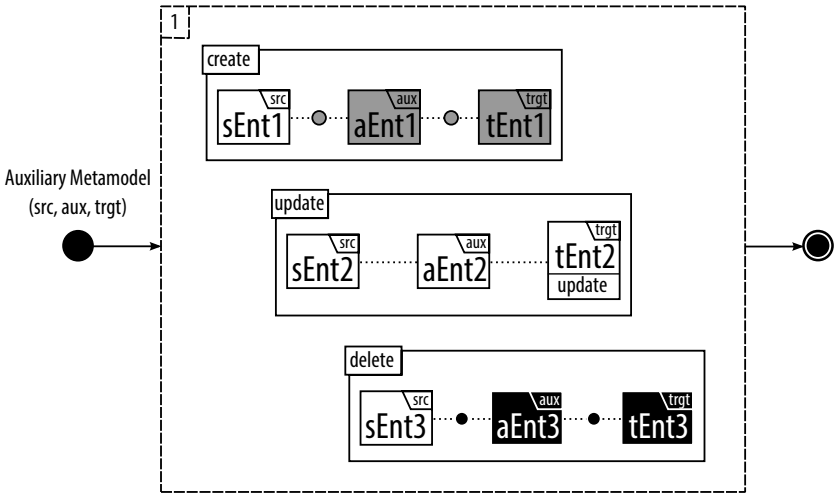

Figure C.31: Auxiliary Metamodel - Structure in DelTa

Appendix C.3.9. Simulating Explicit Rule Scheduling

- **Summary:** This pattern suggests "use of additional application conditions of rules to enforce relative orders of rule execution." [12]

- **Solution:** The solution is depicted in Fig. C.32. 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.

Appendix C.3.10. Simulating Universal Quantification

- **Summary:** The pattern simulates an antedecent "forAll(x|P)" condition by a double negation "not(X|not(P))."

- **Solution:** The solution is depicted in Fig. C.33. 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

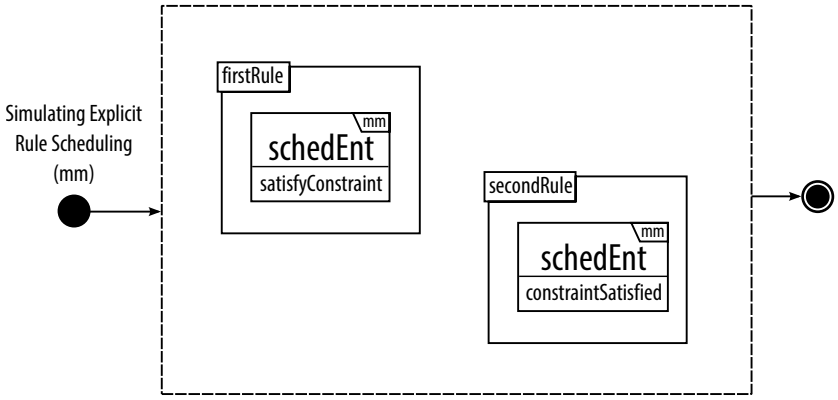

Figure C.32: Simulating Explicit Rule Scheduling - Structure in DelTa

pattern. Finally, the "select" rule tries to select entities those do not satisfy the condition and returns true if it can not find such a rule and vice versa.

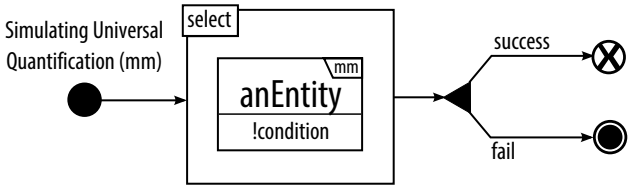

Figure C.33: Simulating Universal Quantification - Structure in DelTa

- **Implementation:** In the "terminatingCondition" rules of Fig. 6, 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 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 negative condition of a firing "transition," which is having less token weight in the incoming edge than tokens of its corresponding "place."