Design Patterns for Model Transformations[*]

Proposal for Thesis Dissertation in Partial Fulfilment
of the Requirements for the Degree of Doctor of Philosophy

Hüseyin Ergin
Computer Science Department
The University of Alabama

June 23, 2014

Abstract

In model-driven engineering, most problems are solved using model transformation. However, the development of a model transformation for a specific problem is still a hard task. The main reason for that is the lack of a development process where transformations must be designed before implemented. As in object-oriented programming, design patterns can benefit "good design" of model transformation tremendously. Hence, this proposal aims to help transformation developers in the design of model transformations through the use of design patterns defined specifically for model transformations. The contributions consist of finding the appropriate language to define model transformation design patterns, identifying design patterns from existing transformation solutions, and generating and detecting design pattern instances for a specific MTL.

[*]Proposal Committee: Eugene Syriani (advisor), Jeffrey Carver, Jeff Gray, Ralf Lämmel, Randy Smith

Contents

1 Introduction

Model-driven engineering (MDE) heavily relies on model transformation. However, although 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. As witnessed in [1], 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. Software developers have been using UML for years to design the problem first. This lets them to see the upcoming problems before the implementation and be able to use these designs to be implemented in different object-oriented languages. One of the most essential contribution to software design was the GoF catalog of object-oriented design patterns [2], which consists of proven solutions to common software engineering problems. Gamma *et al.* used UML class and sequence diagrams to define object-oriented design patterns.

Although there are studies that mention the design patterns in model transformation area [3, 4, 5], they could not be evolved further more. The initial issue these studies face is the lack of a common language to define these model transformation design patterns. The authors often chose a specific model transformation language (MTL) to express the design patterns. This is understandable given the nature of MTLs, which makes the transformation implementation more readable compared to a piece of code written in a general-purpose programming language. However, the need for a common language like UML still remains to make model transformation design patterns more readable and applicable among other MTLs.

The main goal of my thesis is to help model transformation developers in the design of model transformations through the use of design patterns. My contributions consist of finding the appropriate language to define model transformation design patterns, identifying design patterns from existing transformation solutions, and generating and detecting design pattern instances for a specific MTL.

There are several studies to define model transformations independent from the MTL itself [1, 6]. After through analysis of different MTLs and the languages proposed in these studies, I propose DelTa as a candidate language to express model transformation design patterns. DelTa is a neutral and concise language, independent from existing MTLs and only focuses on the essence of model transformations. It is explicitly modeled as a DSL, which means it can be used as input and output in a transformation. Later, this property will help identify design patterns defined in DelTa in existing model transformation solutions. It also has a textual and graphical concrete syntax at the convenience of transformation developers.

Another contribution of my thesis is to provide a collection of model transformation design patterns available to transformation developers. Therefore, I have redefined existing model transformation design patterns [3, 4] in DelTa along with the implementations in different MTLs. This helps the developers both to see the design pattern structure in a suitable language and to see it in practice. I have redefined three design patterns from existing studies and identified two new design patterns. Section 6 shows the efforts to come up with a design pattern candidate by applying similar solutions to three different problems and documents each step of this process, which gives hints about how to identify new model transformation design patterns. Another method of identifying a design pattern is to analyze existing model transformation solutions in different languages. In Section 7, "execution by translation" design pattern is identified by using this method.

The rest of this proposal is organized as follows. Section 2 and Section 3 give some background information on model-driven engineering and design patterns. Section 4 presents the related work about model transformation design patterns, efforts for a language to express them, and identification of design patterns. Section 5 shows the language to define model transformation design patterns, DelTa, and gives the syntax and semantics. Section 6 briefly mentions how to identify a design pattern on a running example step-by-step. Section 7 provides additional model transformation design patterns. Section 8 presents the further work in my plan and the schedule of each work. Finally, Section 9 summarizes what I have done and concludes the proposal. Additionally, there is a list of my published and possible papers during the PhD study in the Appendix A.

2 Background on Model-Driven Engineering

2.1 Model-Driven Engineering

MDE [7] is considered a well-established software development approach that uses *abstraction* to bridge the gap between the problem and the software implementation. MDE uses models to describe complex systems at multiple levels of abstraction. Models are first-class citizens and represent an abstraction of a real system, capturing some of its essential properties. Models are instances of modeling languages which define their abstract syntax, concrete syntax, and semantics. The *abstract syntax* defines the essence of the language, often defined by a *metamodel* The *concrete syntax* defines the graphical or textual representation of the elements of the metamodel. *Semantics* defines the meaning of the language. The static semantics is specified by the metamodel extended with constraints, while the dynamic semantics is often defined by means of a model transformation (either denotational or operational). A model expressed in a modeling language *conforms* to its metamodel. Metamodels themselves are also modeled in a modeling language called metamodeling language, which has a conceptual foundation called *metametamodel*. Transformation, models, metamodels and metametamodels form a four-level architecture in MDE and these levels are called M0, M1, M2, and M3 respectively [8].

In MDE, the core of the development process consists of a series of transformations over models. Typically, a transformation or manipulation is modeled by a *model transformation* that conforms to a specific metamodel. Following Jouault *et al.* [9], the model transformation schema in MDE in Figure 1 illustrates these terms. `Ma` and `Mb` are models that conform to metamodels `MMa` and `MMb` respectively. `MTab`

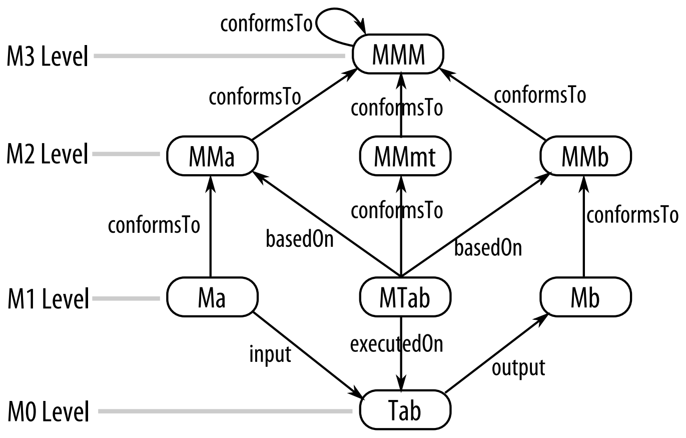


Figure 1: Model transformation schema in MDE.

is a model transformation that conforms to metamodel `MMmt` and takes a model as an input and produces another model as an output. `MTab` is also based on metamodels of both input and output models. All three metamodels, `MMa`, `MMb`, `MMmt` conform to a standard metametamodel.

2.2 Model Transformation

A model transformation is defined as "the automatic manipulation of input models to produce output models, that conforms to a specification and has a specific intent" in [10]. These intents play an important role while creating a model transformation. A model transformation intent is a description of the goal behind the model transformation and the reason for using it [10]. A transformation mainly consists of source and target languages, transformation rules, and scheduling of the rules.

There are roughly two approaches to produce a model transformation [11]: 1) relational 2) graph-transformation-based. Relational approaches usually specify the correspondence between source and target elements, meaning creating the target elements implicitly. Therefore, they focus on a subset of model transformations and have more restrictions on how to create one, *i.e.,* usually the input model is read-only and the output model is write-only. Graph-transformation-based approaches work on graph structures that represent the models and allow more flexibility like a general-purpose programming language. They also allow inplace editing of models, which makes the input and output model the same in the transformation. Graph-transformation-based approaches can have explicit scheduling structure to let developers define when and how the rules will be applied. Relational approaches are straight-forward, therefore this proposal focuses on graph-transformation-based approaches. In the following subsections, the structure of graph-transformation-based model transformation languages are described.

2.2.1 Structure of Graph-transformation-based Languages

Transformation Rules are the smallest units of a model transformation. A transformation rule has many different features according to [11]. The *domain of a rule* defines how a rule can access elements of models. A rule is a declarative construct that dictates *what* shall be transformed and not *how*. It consists of pre-condition and post-condition patterns. The pre-condition pattern determines the applicability of a rule: it is usually described with a left-hand side (LHS) and optional negative application conditions (NACs). The LHS defines the pattern that must be found in the input model to apply the rule. The NAC defines a pattern that shall not be present, inhibiting the application of the rule. Constraints can be also be specified over the attributes of LHS and NAC pattern elements. The right-hand side (RHS) describes the post-condition pattern that must be found in the output model after the rule is applied. Imperative actions can be also be specified over the attributes of RHS pattern elements. An advantage of using the rule-based transformation paradigm is that it allows to specify the transformation as a set of operational rewriting rules instead of using imperative programming languages. A rule example with LHS, RHS and NAC parts is depicted in Figure 2. This rule is taken from a model transformation that translates an UML

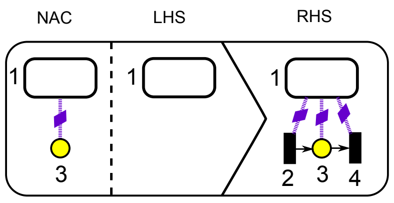


Figure 2: A sample model transformation rule.

activity diagram model to a behaviorally equivalent Petri net model [12]. The rule can be read as "if an activity (labeled 1) that is not associated with a place (labeled 3) is found, then create a place and two transitions (labeled 2 and 4), and relate them with temporary trace links". This rule has a *graphical syntax* using elements from the concrete syntax of the source and target domains (activity diagrams and Petri net).

Rule Scheduling is an important phase in the development of a model transformation. Scheduling mechanisms determine the order in which individual rules are applied [11]. One can distinguish between implicit and explicit scheduling. When the scheduling of a transformation language is *implicit*, the modeler has no direct control over the order in which the transformation units are applied. On one hand, a transformation language can be *unordered*, i.e., it simply consists of a set of rules. In this case, the order of application of the rules is entirely determined at run-time. It completely depends on the patterns specified in the rules. Applicable rules are selected non-deterministically until none apply anymore. The scheduling of a language can be *explicitly* specified by the modeler. In explicit internal transformation languages, a rule may explicitly invoke other rules. For example in ATL [9], a matched rule (implicitly scheduled) may invoke a called rule in its imperative part. Finally, in an explicit external transformation language, there is a clear separation between the rules and the scheduling logic. Ordered transformations specify a control mechanism that explicitly orders rule application of a set of rules. Examples are: priority-based, layered/phased, or with an explicit workflow structure. Most transformation languages are partially ordered, however. That is, applicable rules are chosen non-deterministically while following the control specification.

2.2.2 Model Transformation Languages

There are many model transformation languages in the literature. Some examples are Henshin [13] from Arendt *et al.*, which is a language that operates on models in Eclipse Modeling Framework (EMF) and has visual syntax, editing functionalities, execution and analysis tools; GReAT [14] from Agrawal *et al.*, which consists of three distinct parts; pattern specification language, graph transformation language and control flow language; FUJABA [15] from Klein *et al.*, which is one of the first tools to do code generation from UML models and UML model generation from code; Viatra2 [16] from Varro and Balogh, which provides a rule and pattern-based language for manipulating graph models by using graph transformation and abstract state machines; and AGG [17] from Taentzer, which lets graphs to be attributed by Java objects and equips graph transformation with computations on these objects.

Each of these languages have a unique set of structure combinations (e.g., different rule and scheduling structure, different directionality). Jouault and Kurtev [18] compared a number of model transformation languages in terms of transformation scenarios, paradigm, directionality, cardinality, traceability, query language, rule scheduling, rule organization and reflection.

2.3 MoTif

MoTif is a graph-transformation-based model transformation language and a short name for "Modular Timed Graph transformation". MoTif and its semantics is based on the T-Core [19]. It introduces an explicit notion of time and allows to model the interruption for every rule in the execution.

Figure 2 depicts an example rule in MoTif. The rule is part of another study [12] and basically add some new petrinet elements to UML activity diagram action nodes while preventing the rule to be applied more than once by adding a NAC and maintaining traceability links to the original nodes. The rule consists of three parts. The first part is NAC and separated with dashed line from other parts. Multiple number of NACs can be added to a rule. The second part is LHS on the left of the big arrow. That is the precondition pattern to be found in the input model. The third part is RHS on the right of the big arrow and the post-condition pattern to be applied to the model. As one can easily realize, concrete syntax can be used in rule designing phase.

Rule scheduling in MoTif is explicitly defined by another structure, which is also modeled. The structure allows to define what happens when the rule is matched or not. A sample scheduling sequence is depicted in Figure 3. The rules are single lined green boxes and they have input ports and output ports

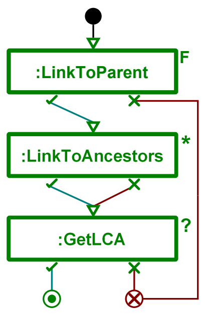


Figure 3: Scheduling of MoTif Rules.

for success and failure. Success case is finding the match in the input. All output ports can be connected to any other rule's input ports or the output ports of the block.

MoTif consists of rule blocks [20]. Each rule block can be either atomic or composite. Some of the atomic rule block can be found in Figure 3 and are listed below. The content of these rule blocks appears in

- *ARule:* means a regular atomic rule. It is a simple rule that is executed only once.
- *FRule:* means 'For all Rule'. The matches are found for the input model and this rule is applied to all found matches. For example in Figure 3, rule LinkToParent is an FRule.
- *SRule:* means 'Star Rule' and applies the transformation to all matches as long as the rule is applicable. Therefore it is applied to the resulting model cumulatively after each application. For example in Figure 3, rule LinkToAncestors is an SRule.
- *QRule:* means 'Query Rule' and mostly consists of only LHS and NACs For example in Figure 3, rule GetLCA is an QRule.
- *CRule:* means 'Composite Rule' and can refer to another full MoTif transformation.

Composite rule blocks allow one to encapsulate the composition of rule blocks. Some of them express flow structures, such as branching and looping.

T-Core Under MoTif, there is T-Core [21], which stands for "Transformation Core". It is a collection of primitive operators for model transformation. T-Core offers the following eight primitives:

- *Matcher* finds all possible matches of the condition on the graph embedded. After matching, it stores all the matches in the packet.
- *Rewriter* applies the required transformation on the match specified in the packet it received.
- *Iterator* chooses a match among the set of matches of current condition of the packet. The match is chosen randomly and choosing the match continues until a maximum number is achieved.
- *Resolver* resolves a potential conflict between matches and rewritings by prohibiting any changes to other matches in the packet.
- *Rollbacker* is used as a recovery point that allows backward recovery of packets.
- *Selector* is used when a choice needs to be made between multiple packets processed concurrently by different constructs.

- *Synchronizer* is used when multiple packets processed in parallel need to be synchronized.
- *Composer* is a modular encapsulation of the elements and any other primitives can be added to the encapsulation.

The rules of MoTif can be expressed in terms of these primitives. Additionally, any MTL can be mapped to T-Core and is executed in AToMPM as described in the following paragraph.

Implementation of MoTif and T-Core in AToMPM AToMPM [22] is a web-based modeling environment for designing domain-specific modeling language environments, perform model transformations, manipulating and managing models. Figure 4 depicts what has been done to execute MoTif and T-Core transformations in AToMPM. First, I created a DSL for MoTif. A transformation in AToMPM is

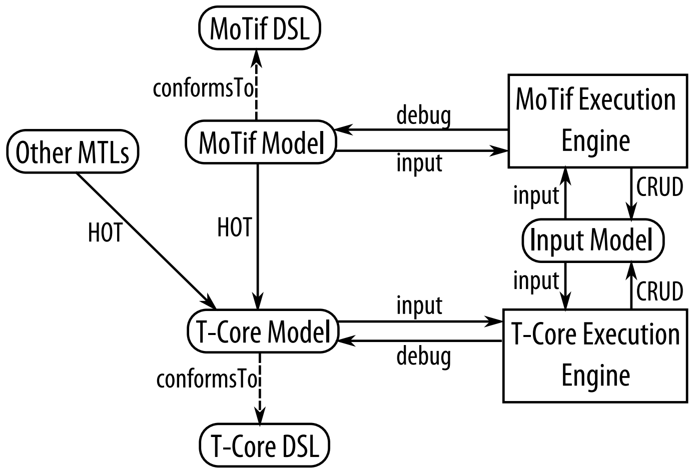


Figure 4: Overall diagram of implementation.

a model conforming to this DSL. Then I implemented the MoTif execution engine in Python. The engine accepts a MoTif transformation and a model as inputs, and executes the transformation step-by-step. It manipulates the input model with CRUD operations (creation, read, update, deletion of model elements) according to the result of the transformation. The engine can also debug MoTif models by highlighting the current rule. The engine has two modes: 1) executing the whole transformation and showing the resulting model at the end 2) executing the transformation step-by-step and reflecting the changes at each step. This gives developers the option to see the effect of each rule in the transformation. I also developed similar engine for T-Core. This allows any MTL to be executed in AToMPM by defining a higher-order transformation (HOT) from the MTL metamodel to T-Core. Since T-Core provides the primitives of a graph-transformation-based language, each MTL will have an execution engine regardless of it is supported by AToMPM, or not. I tried this approach by implementing a HOT from MoTif to T-Core and executing a MoTif transformation using the T-Core engine. I also implemented a HOT from another domain specific MTL created from scratch to T-Core and sucessfully executed the transformation.

3 Background on Design Patterns

Design patterns are reusable structures that can help to overcome any problem to be solved from scratch. Each pattern "describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, is such a way that you can use this solution a million times over, without ever doing it the same way twice" [2]. Design patterns emerge from reusable idioms found across different languages through encapsulation and abstraction. There are many studies involving hundreds of design patterns in the literature. Each of them describe a solution to some kind of a problem. The use of design patterns leads to the construction of well-structured, maintainable and reusable software systems [23].

There are several design pattern studies in the literature for different areas. Design patterns are listed to be used in software architecture [24], building Corba applications [25], real-time systems [26], distributed computing [27], and embedded network systems [28]. Also, there are design pattern studies for different parts of model-driven engineering. In [29], Cho and Gray proposed a list of metamodel design patterns for different problems faced during metamodel design. There are also design patterns in model transformation [3, 4], which I investigate in details.

In this section, I focus on object-oriented design patterns given its popularity and acceptance in the community.

3.1 Object-oriented Design Patterns

Object-oriented pattern cataloging process began as a part of Erich Gamma's PhD thesis [2]. Then the other authors joined the process and design patterns found the last appropriate structure to be published as a book. There are now 23 standard object oriented design patterns in the book. Actually before Gamma *et al.* [2]'s work, there were still programming languages that use design patterns without mentioning the strict name. For example model-view-controller structure in Smalltalk-80 is an earlier example of a design pattern [30].

3.1.1 Structure

The essential elements of design patterns are explained in Gamma *et al.* [2]. These are the four main primitive elements under a design pattern and basically used to express a design pattern's purpose and results. They are listed as follows:

- The **pattern name** is actually a handle to summarize all other fields in the design patterns essential elements. It lets developers to freely talk and understand the design as an abstraction. Finding a good name is one of the hard parts of developing a pattern.
- The **problem** describes when to apply the pattern. Mostly the problem and its context are explained in this field. These may include specific design problems, class or object structures and a list of conditions that must be met before application.
- The **solution** describes the elements that are parts of the design, their relationships, responsibilities and collaborations. Since a pattern is like a template, it has to provide a solution to be applied in many different situations. The solution is generally given with UML class diagrams.
- The **consequences** are the results and trade-offs of applying the pattern. These are critical for evaluating design alternatives and for understanding the costs and benefits before applying the pattern. Language and implementation issues, impacts on a system's flexibility, extensibility and portability may also help users to understand and evaluate design patterns.

Other than these elements, there are more fields in [2] to describe a design pattern. These are; pattern name and classification, intent, also-know-as, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses, related patterns. *Intent* is an important field both in design patterns and model transformations that needs to be mentioned more. This field is the key to select the right design pattern for a problem and the right constructs for a model transformations. Although the rest of the fields are not crucial for a design pattern, they help to explain it better and precise.

Hasheminejad and Jalili [31] also introduce two main sections to categorize these fields: problem domain and solution domain. The *problem domain* describes the problem context where the pattern can be applied and has these fields: intent, motivation and applicability. The *solution domain* describes the structure and collaborations of the pattern solution being applied to the problem and has these fields: structure, participants, collaborations, consequences, implementation and related patterns.

3.2 Limitations of Design Patterns

Design patterns are accepted to be a useful structure in the aim of re-usability and readability. They have some advantages and disadvantages. A design pattern is based on the problem, context and constraints. Therefore it doesnot aim for all problems.

The goal of design patterns is to increase quality metrics to satisfy some quality criteria. This often comes with a trade-off. In design patterns, re-usability criteria mostly conflicts with efficiency criteria. For example, visitor pattern lets people traverse class structure and process on them in an efficient way, while it requires nearly double number of new classes created. Also, it is not clear how many design patterns are enough in a project. There is a probability that you mess up the code by applying too many unnecessary design patterns [32].

Another point, applying design pattern is not automated yet and it still depends on the manual decision of the designer. Hasheminejad and Jalili [31] proposed an automatic two-phase method for design pattern selection, but this does not reduce the impact of the designer in selection process. Blomqvist [33] proposed a pattern selection approach by ranking the design patterns onthologically and matching them with terms.

However, re-usability, readability and maintainability are so important criteria that this makes design patterns always popular.

4 Related Work

4.1 Model Transformation Design Patterns

The first work that proposed design patterns for model transformation was by Agrawal *et al.* [3]. They defined the *transitive closure* pattern to create traceability links between the parents and ancestors of the elements. The *leaf collector* pattern traverses a hierarchical tree to find and process all leaves. This can be considered as an application of the visitor pattern in Section 7.3 where the `visitEntity` rule is only applied on leaves. The *proxy generator* idiom is not a general design pattern, since that it is specific to languages modeling distributed systems where remote interactions to the system need to be abstracted and optimized.

Iacob *et al.* [4] defined five other design patterns for outplace transformations. The *mapping* pattern dictates to first map entities and then relations. Since it is described using QVT-R, we consider it as an implementation of our ER mapping pattern. The *refinement* pattern proposes to transform an edge into a node with two edges in the context of a refinement so that the target model contains more detail. The *node abstraction* pattern abstracts a specific type of node from the target model while preserving the original relations. The *flattening* pattern removes the composition hierarchy of a model along by replacing the containment relations. We plan to generalize these three patterns and define them in DelTa. The *duality* pattern is not a general design pattern, since it is specific to languages for data control flow modeling by changing by converting edges to nodes and vice versa.

Bézivin *et al.* [5] mined ATL transformations and ended up with 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 and therefore this pattern is obsolete now.

Levendovszky *et al.* [34] proposed domain-specific design patterns for model transformation as well as other DSLs. In their approach, they defined design patterns with 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.

The first issue with these previous works is that all the design patterns are defined using GReAT, QVT-R, ATL, and VMTS respectively. Therefore, they should not be considered as design patterns for model transformation, but as implementations of design patterns in a specific MTL. The second issue is that 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 MTLs.

4.2 Model Transformation Design Pattern Languages

Lano *et al.* [6] proposed other useful patterns using UML class diagrams and OCL constraints (first-order logic) to specify model transformations. Each transformation is described with a set of *assumptions* that represent the precondition of a rule, *constraints* that represent the postcondition of a rule, *ensures* for additional constraints, and *invariants*. The design patterns are for exogenous transformations only. The *conjunctive-implicative form* pattern dictates to separate the creation target entities that are at different hierarchical levels into different phases. For example, the *map objects before links* pattern, essentially our ER mapping pattern, is an instance of this generic pattern. Another instance of this pattern is the *recurrent constraints* pattern where the creation of a target entity may require a fixed point computation. Two other instances of the conjunctive-implicative form pattern are the *entity splitting* and *entity merging* patterns that essentially correspond to the one-to-many and many-to-one variants of the ER mapping pattern respectively. The *auxiliary metamodel* pattern suggests to use an auxiliary metamodel when the mapping from elements of one language to another is too complex.

In Lano *et al.*'s approach, the choice of the design pattern language hinders the understandability of the patterns. This also makes them hard to implement in MTLs other than UML-RSDS. Additionally, they defined implementation patterns. In contrast with design patterns, they are guidelines to implement the assumptions and constraints of transformation specifications in a MTL. The description is done on an abstract implementation language that supports sequencing, branching, looping and operation calls.

Guerra *et al.* [1] proposed a collection of languages to engineer model transformations and, in particular, for the design phase. Rule diagrams (RD) are used to describe the structures of the rules and what they do in the low level implementation phase. RD is defined at a level of abstraction that is supposed to be independent from existing model transformation languages. But its purpose is to generate a transformation rather than to define design patterns. However to generate a transformation, RD relies on different rule and mapping diagram instances for different model transformation languages. In RD, rules focus on mappings rather than constraints and actions. Hence, they specify designs for both unidirectional and bidirectional rules. The execution flow of RD supports sequencing rules, branching in alternative paths based on a constraint, or non-deterministically choosing to apply one rule. They also allow rules to explicitly invoke the application of other rules. RD is inspired from QVT-R and ETL and is therefore more easily implementable in these language.

4.3 Identification and Detection of a Design Pattern

Design pattern detection is also another field that can help to identify model transformation design patterns. Since detection techniques are mostly to find software design patterns that are mentioned in [2] and [26], they need to be modified to fit in the model transformation world. There are many design pattern detection techniques in the literature.

Dong *et al.* [35] studied a comprehensive review on these techniques. Each design pattern is generally described from different perspectives. These are the distinguishing characteristics of design patterns and divided into three: structural, behavioral and semantic characteristics. *Structural* aspect is relatively easy and can be detected from source code or architectural system design and mostly based on the relationships between classes, such as generalization, association, aggregation. *Behavioral* aspect is typically described by method invocations. These invocations can be checked in a static way or a dynamic way at system run time. *Semantic* aspect is defined in different ways in literature but it basicly refers to the semantic meaning of some entities in the system. One can take advantage of naming conventions, programming guidelines, multiplicities.

Some detection approaches take the reverse engineering tools into consideration to get an intermediate representation of the system and design pattern discovery is done on these *intermediate representations* instead of source code. These efforts are usually done to reduce the search complexity. There are several common intermediate representations such as Abstract Syntax Tree, Abstract Semantic Graph, bit vector, matrix.

Design pattern detection techniques differ from each other in some other criteria as well. Another criteria is *exact matching* or *approximate matching*. Most approaches search a piece of architectural design that structurally confirms to the structural characteristic of the pattern. Some patterns can't be matched perfectly. Therefore, approximate matching is used in some approaches. Also final discovery results also presented differently. Design patterns discovered by using a technique are generally visualized in the UML diagrams or class tree hierarchy.

Apart from these criteria, most approaches provide tools to automate the detection process. Some may require user interaction as well. The detection of each approach generally supports a subset of design patterns.

5 A Language for Model Transformation Design Patterns

One possible idea that can be deduced from the related works is the lack of a common design pattern language to express model transformation design patterns. The benefits of having such a language are to facilitate, understand, document, communicate, and reason about patterns in a standard way [36]. Also, the language should be independent from regular model transformation languages (MTL). Object-oriented design patterns are expressed in UML which is independent from general-purpose programming languages. I could have used an existing MTL as a notation, however this can lead developers to think the language itself is a transformation language and executable. The need is a notation that expresses how elements within a rule are related and how rules are related with each other. Therefore, the language should be I propose DelTa as a language for model transformation design patterns.

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. In this respect, DelTa offers some concepts borrowed from any MTL, 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.* [2] used UML class, sequence and state diagrams together to define design patterns for object-oriented languages. In the following, I describe the abstract syntax, concrete syntax, and informal semantics of DelTa.

5.1 Abstract Syntax

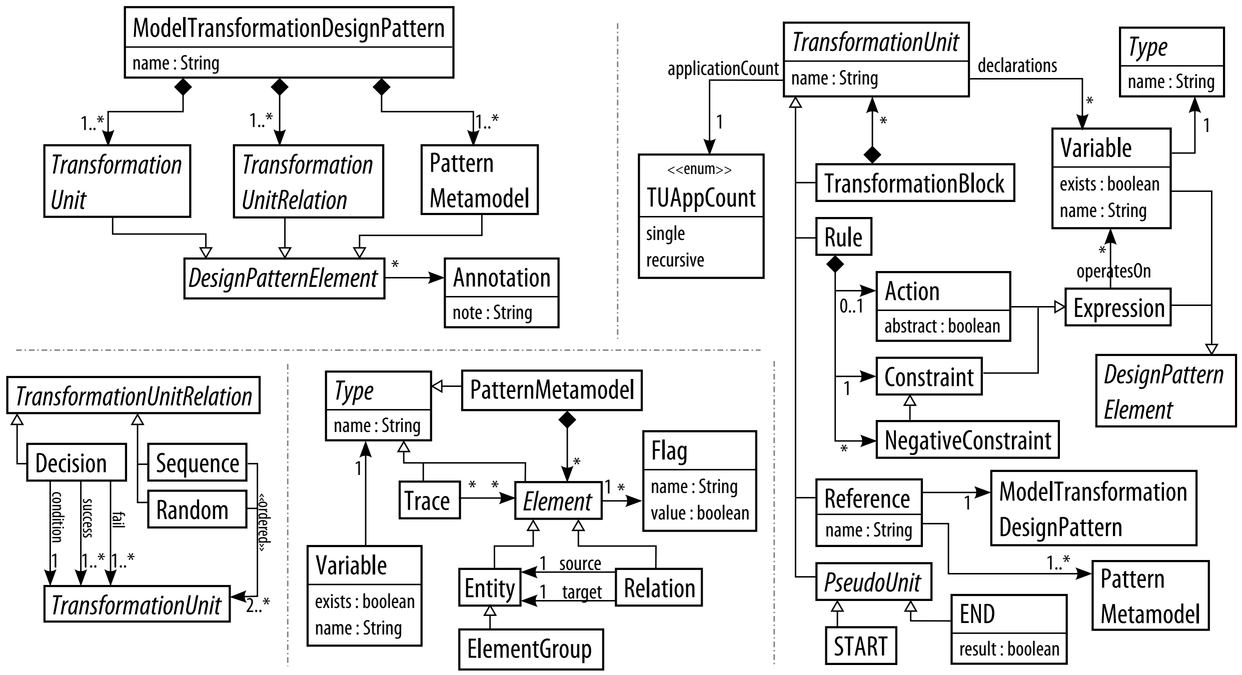


Figure 5: DelTa Metamodel

As depicted in Figure 5, 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 [20]. TUs represent the concept of rule in graph-based model transformations [37]. With the reference TU, a design pattern can refer to another complete design pattern by passing the pattern metamodel as parameter. A MTDP rule consists of a `constraint`, an `action`, and optional `negative constraints`. These correspond to the usual left-hand side (LHS), right-hand side (RHS) and negative application conditions (NACs) in graph transformation. A constraint defines the pattern that must be present, a negative constraint defines the pattern that shall not be present, and the action defines the changes to be performed on the constraint (creation, deletion, or update). All these expressions operate on strongly typed `variables`.

There are three types for variables: a `pattern metamodel`, a metamodel `element`, or a `trace`. The pattern metamodel is a label to distinguish between elements from different metamodels, since a MTDP is independent from the source and target metamodels used by an actual model transformation. When implementing a MTDP, the pattern metamodel shall 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 [38]. The metamodel labels also indicate the number of metamodels involved in the transformation to be implemented. Metamodel elements are typically either entity-like and relation-like elements, this is why it is sufficient to only consider `entities` or `relations` in DelTa. An element may be assigned boolean `flags` to refer to the same variables across rules. Undeclared flags are defaulted to `false`. This is similar to pivot passing in MoTif and GReAT, and parameter passing in Viatra2. When implementing a MTDP, flags may require to extend the original or ramified metamodels with additional attributes. An `element group` is an entity that represents a collection of entities and relations implicitly, when fixing the number of elements is too restrictive. Traceability links are crucial in MTLs but, depending on the language, they are either created implicitly or explicitly by a rule. In DelTa, I opted for the latter, which is more general, in order to require the developer to take into account traceability links in the implementation.

As surveyed in [19], 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 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 which are two TU `application counts` in DelTa. All other flavors of TUs can be expressed in TURs as demonstrated in [19]. For reuse purposes, rules in DelTa can be grouped into `transformation blocks`, similarly to a `Block` in GReAT.

As surveyed in [39, 11], in any MTL, rules are subject to a scheduling policy, whether it is implicit or explicit. For example, AGG uses layers, MoTif and VMTS [40] use a control flow language, and GReAT defines causality relations between rules. As shown in [21], 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 three TURs of DelTa: `Sequence`, `Random`, and `Decision` that are explained in Section 5.3. The former two act on at least two TUs and the latter has three parts; condition, success and fail TUs. `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 on the particular design pattern element. This is especially used for element groups and abstract actions.

5.2 Concrete Syntax

Listing 1 shows the EBNF grammar implemented in Xtext.

Listing 1: EBNF Grammar of DelTa in XText

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

ElementGroup: 'ElementGroup' ELEMENTNAME (',' ELEMENTNAME)* ;
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? ;
Action: ('abstract 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' | 'Random') ':' ;
```

The structure of a DelTa design pattern is as follows. A new design pattern is declared using the *mtdp* keyword. This is followed by a list of metamodel names. The rules are defined thereafter. Rules can be contained inside transformation blocks represented by the *tblock* keyword. The '∗' next to the name of the rule indicates that the rule is recursive; the application count is single by default. Since reference is also a TU, it is defined at this level. A rule always starts with the declaration of all the variables it will use in its constraints and actions. Then, the *constraint* pattern is constructed by enumerating the variables that constitute its elements. Elements can be prefixed with '∼' to indicate their non-existence. Flags can be defined on elements using the square bracket notation. Optional negative constraints can be constructed, followed by an action. An abstract action may not enumerate elements. The final component of a MTDP is the mandatory TUR definitions. A TUR is defined by its type and followed by a list of rule or transformation block names. As an exception, decision TUR is a single line conditional that creates a branch according to the success or fail of the condition rule. Annotations are enclosed within '#'. In [41], all design patterns are depicted with their textual syntax.

I opted for a textual concrete syntax for DelTa at first. However, after informally surveying the model transformation community, I discovered that some design patterns, such as "visitor" or "fixed point iteration", that require more complex scheduling of the rules are difficult to understand in textual syntax. Therefore, I opted for an alternative graphical syntax for DelTa that is equally expressive as its textual counterpart. Figure 6 illustrates the graphical concrete syntax of all metamodel elements.

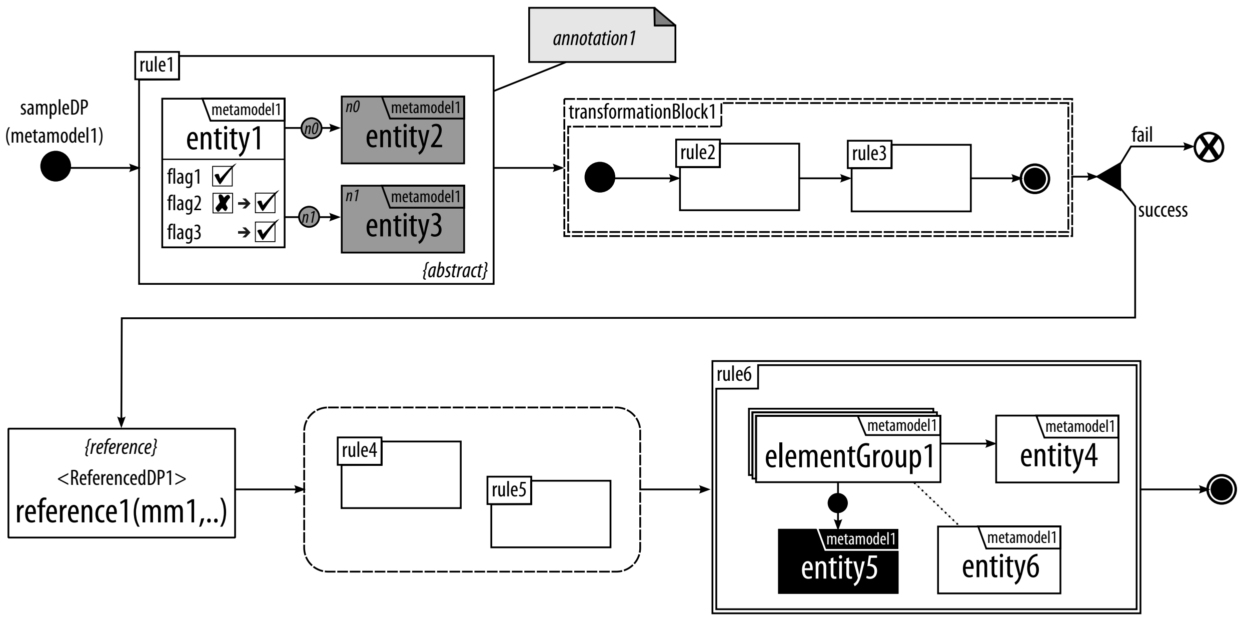


Figure 6: A dummy design pattern illustrating the graphical syntax of DelTa.

DelTa design patterns starts with a **Start** node that is characterized by the name of the design pattern and the pattern metamodels involved. It is represented with a filled circle. The **End** node represents the end of a design pattern with a result of success or fail depicted with a cross or an inner black circle respectively. DelTa design patterns focus on the relations between the TUs. Each TU contains the pattern metamodel elements and the rule description. "rule1" is a **rule** to be applied once, whereas "rule6" is applied recursively. Optionally, rules can be marked abstract meaning that the rewriting part is left to the implementation. “transformationBlock1” is a **block** that represents a nested hierarchy of other TUs. TUs are connected with with arrows which represents the **Sequence** of the order of their application. The outcome of a TU is either success or fail. If this differentiation matters to the pattern, a **Decision** node is used, represented with a triangle of one input port and an output port for each outcome. The sequence chooses which way to follow according to the result of the previous rule or block. “reference1” is a **reference** to another DelTa design pattern named “ReferenceDP1”. When referencing a design pattern, passing the information of the metamodels to the referenced design pattern is required. They appear between the parenthesis after the reference’s name. “rule4” and “rule5” are inside a **Random** TUR, which means that at most one of them will be applied at random.

Each rule describes the CRUD operations on elements from pattern metamodel. The name of the pattern metamodel appears on top right of any element (*e.g.,* to distinguish which metamodel the element belongs). “entity1” is an **entity** with three **flags**. Flags can be true (“flag1”) or false (“flag2”), created (“flag3”) or modified (“flag2”). White elements are to be matched, gray elements are to be created and black elements are to be deleted. To specify an element group of unknown number, there is the **Element-Group**. It is represented with a group of elements stacked such as the “elementGroup1” in the “rule6”. Elements and element groups are connected to each other with `Relations` or `Traces`. A **Relation** is an arrow, for example the “elementGroup1” is connected to “entity4” with a relation. A **Trace** is a dotted line, for example the “elementGroup1” is connected to “entity6” with a trace. A rule consists of **constraints**, **negative constraints**, and **actions**. Negative constraints are marked with “n” and some number, for example “entity2” has the negative constraint “n0” on top left of its icon.

“rule1” is to be read as follows. An entity “entity1” from “metamodel1” must be matched with flags “flag1” true and “flag2” false. Furthermore, there should not be any relation between “entity1” and two other entities. Then, two entities must be created (“entity2” and “entity3”), “flag2” must be set to false, and a new “flag3” must be initialized to true. “rule6” is to be read as follows. If an entity group is related with two entities (“entity4” and “entity5”) and shares a trace with another third entity, then one of the related entities shall be deleted. Finally, each element in DelTa can be **annotated** with a note, inspired from UML.

5.3 Informal Semantics

The semantics of MTDP rules is borrowed from graph transformation rules [37], but adapted for patterns. Informally, a MTDP rule is applicable if its constraint can be matched and no negative constraints can. If it is applicable, then the action must be performed. Conceptually, we can represent this by: $constraint \wedge \neg neg1 \wedge \neg neg2 \wedge \ldots \rightarrow action$. 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 [42]. Elements present in the action must be created or have their flags updated. Negated variables in an action indicate the deletion of the corresponding element. Only abstract actions are empty, giving the freedom to the actual implementation of the rule to perform a specific action. Flags are not attributes but label some elements to be reused across rules.

MTDP rules are guidelines to the transformation developer and are not meant to be executed. On one hand, the constraint (together with negative constraints) of a rule should be interpreted as *maximal*: *i.e.,* a MT 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 MT 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 MT 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 the rule is side-effect free, meaning that it cannot perform any modifications.

The scheduling of the TUs of a MTDP (or contained inside a `transformation block`) must always begin with `START` and end with `END`. TUs can be scheduled in four ways. The `Sequence` relation defines a sequencing relation between two or more TUs regardless of their applicability. For example `Sequence:A,B` means that `A` should be applied first and then `B` can be applied. The `Random` relation defines the non-deterministic choice to apply one TU out of a set of TUs. For example `Random:A,B` means that `A` or `B` should be applied, but not both. The `Decision` relation defines a conditional branching and applies the TUs in the success or fail branches according to the application of the rule in the condition. For example `A?B:C` means that if `A` is applicable then `B` should be applied after, otherwise `C` should be applied. Note that the latter TUR can be used to define loop structures. For example, `A?A:A` is equivalent to defining `A` as recursive, *i.e.*, `A*`. The notion of applicability of a transformation block is determined by the result of its `END` TU. For example, consider a transformation block `T` and a rule `R` and `P`. The scheduling `T?R:P` means that if `END[result=true]` is reached in `T`, then `R` will be applied. The graphical concrete syntax explain all these scheduling tricks better.

6 Identification of a Model Transformation Design Pattern

The identification of a model transformation design pattern is not an easy task. It requires solving and analyzing some solutions to finally come up with a design pattern. In this section, I propose to solve the well-known lowest common ancestor (LCA) problem [43] using model transformation. For this purpose, I solve the problem using a naïve and an improved solution. I show that the latter improves the quality metrics of the model transformation with respect to efficiency criteria. Then, I identify two other problems that can be solved using an approach similar to the improved LCA solution. I therefore generalize the solution to a design pattern as it describes a solution for recurrent problems and increases the quality of the model transformation that implements it.

6.1 Running Example

LCA is a general problem in graph theory and is typically defined over a directed tree structure. Essentially, it attempts to find the lowest shared ancestor between two given input nodes of the tree. For example in Figure 7, the LCA of nodes D and J is node A. In this instance, one can compute the LCA of

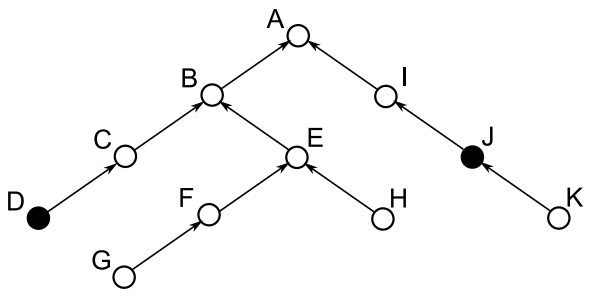


Figure 7: Tree instance for LCA problem

node D and node J to be node A.

6.1.1 Naïve Solution

Typically, solutions using model transformation approaches tend to take advantage of the declarativeness and non-determinism of rule-based systems. In the first solution proposed, I first create all ancestor links of every node as depicted by the first three rules in Figure 8. Then `GetLCA` rule checks if, given the two initial nodes (A and B), there is an ancestor node common to both nodes that do not have a successor that is also a common ancestor of the two nodes. The rules and scheduling of these rules are depicted in Figure 8. For this study, I have focused on three metrics: *the number of rule applications* counts how

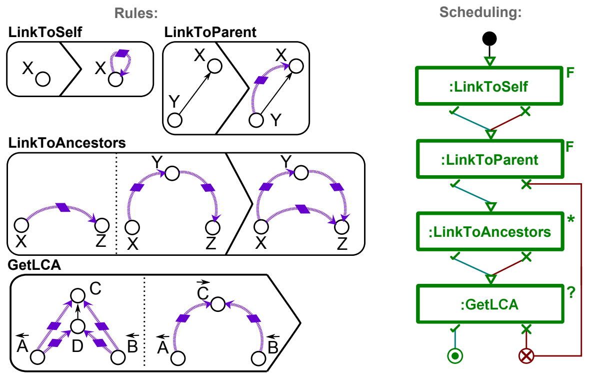


Figure 8: Rules for naïve solution

many times the rule is applied, *the size of the rule* counts the number of elements present in the patterns of each rule, and *the number of auxiliary elements created* counts the number of ancestor links created to compute the LCA.

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

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

After all ancestor links are created, the `GetLCA` rule is applied only once and returns the LCA of the given input nodes if it exists. The NAC part of the `GetLCA` rule guarantees that the solution is the lowest one among other common ancestors. The metrics for the naïve solution are depicted in Table 1.

6.1.2 Improved Solution

In the improved solution, I use locality, focusing on only the given input nodes. I adopt an iterative approach. I start to create ancestor links one step at a time and, at each time, I check for a solution. The rules and scheduling of these rules are depicted in Figure 9.

The `LinkToSelf` rule creates self-ancestor links for the given input nodes only and therefore is applied twice. To acheieve that, I use the pivot feature in MoTif which forces the rule to be applied on bound or elements. That is, A and B are parametrized nodes bound to nodes from the input model at run-time. Then, the `LinkToParent` rule creates ancestor links to the parents of input nodes, which is applied twice.

Table 1: Metrics for naïve and improved LCA solutions

Rules	**Size of rules**		**# Rule Applications**		**# Auxiliary Elements**	
	Naïve	***Improved***	***Naïve***	***Improved***	***Naïve***	***Improved***
LinkToSelf	3	3	n	2	$n-1$	2
LinkToParent	7	7	$n-1$	2	$O(n^2)$	$2n-2$
LinkToAncestors	14	14	$O(n^2)$	$2n-2$	$O(n^2)$	$2n-2$
GetLCA	14	14	1	n	0	0
Total	**38**	**38**	$\mathbf{O(n^2+2n)}$	$\mathbf{3n+2}$	$\mathbf{O(n^2+2n)}$	$\mathbf{2n+2}$

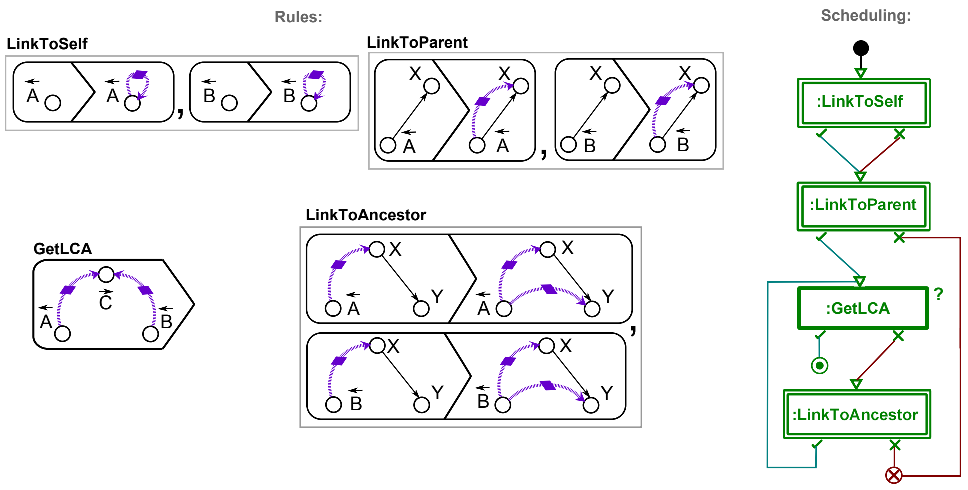


Figure 9: Rules for improved solution

This results in an intermediate form of the tree instance, which may possibly solve the LCA task. Therefore, I apply the `GetLCA` rule and try to find the solution if it exists. If I cannot find a solution, I execute the `LinkToAncestor` rule and create one more level of ancestor links. Again, I use only the given input nodes. With only one more step, this rule takes the intermediate form closer to a solution. Then, I use the `GetLCA` rule to check again. These iterative steps continue until the `GetLCA` rule finds a solution or the `LinkToAncestor` rule fails by not making a contribution to the solution *i.e.*, if the root is reached and `GetLCA` fails. For the tree instance in Figure 7, the solution is found in three steps. Therefore, the `GetLCA` rule is applied four times and the `LinkToAncestor` rule is applied three times. In general, the given input nodes might be in different depth levels (k_1 and k_2 respectively). The ancestor link creation continues up to the root node, so the maximum of depth levels is the number of iterations needed to find the solution. In the worst case, this depth can be n and I create $n-1$ ancestor links. Therefore, the `LinkToAncestor` rule is applied a total of $2(n-1)$ times for input nodes and the `GetLCA` rule is applied n times.

Metrics for the improved solution are also depicted in Table 1. One can clearly see the improvement by comparing the metric counts between naïve solution and improved solution. All three metrics are related to the efficiency quality criteria. Therefore, I can say the improved solution is more efficient than the naïve solution. I did not take the execution time of the model transformations because they are already proportional to the enumerated metrics.

6.2 Similar Problems

In this section, I identify and solve two more problems from very different domains using model transformation.

6.2.1 Equivalent Resistance

In electrical circuits, the computation of the equivalent resistance of the whole circuit is a common task. Finding the equivalent resistance in a series of connected resistors is an interesting problem to apply our design pattern. In this case, the transformation takes as input an electrical circuit model with resistors connected both in serial and parallel. The rules are depicted in Figure 10. The `IsFinished` rule looks for resistors set in serial or parallel in the circuit. If the rule cannot find any more serial or parallel resistors, it will return the single resistor as the equivalent resistance. The `CalculateUnitEquivalentResistance` rule calculates equivalent resistance for only a set of serial and/or parallel resistors and directs the control flow to the `IsFinished` rule again depicting a loop.

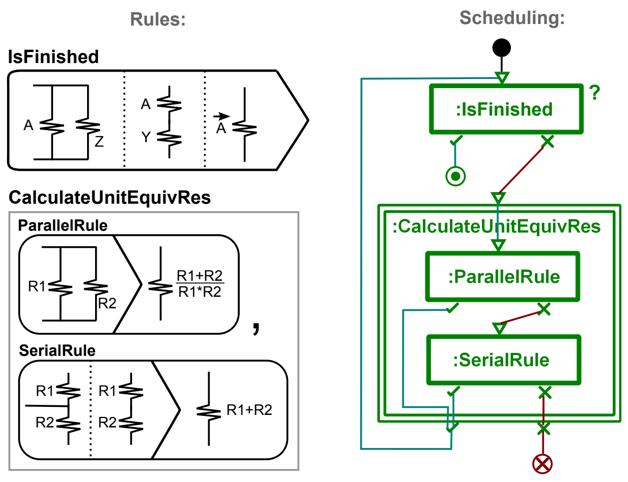


Figure 10: Rules for Equivalent Resistance Problem

6.2.2 Dijkstra’s Algorithm for Shortest Path

Dijkstra’s algorithm is a well-known graph search algorithm that returns the shortest path and length of this path between two nodes, source and target. The solution is provided in Figure 11. The input model is a directed and weighted tree. The `VisitImmediateNeighbors` rule initiates the algorithm by visiting the immediate neighbors of the source node. After a visit, each node is assigned with the weight of the path and is colored in red to represent that it is visited. The terminating criteria of the algorithm is visiting all nodes, which is ensured by the `IsAllNodesVisited` rule. If there are still unvisited nodes, then the `VisitOneMoreHop` rule is executed. The `VisitOneMoreHop` rule selects the smallest number of weighted nodes among visited ones and calculates the new weights for the unvisited neighbors of this node. After each node is visited, the target node will have the length of the shortest path as value and the path with purple marked arrows will be the shortest path.

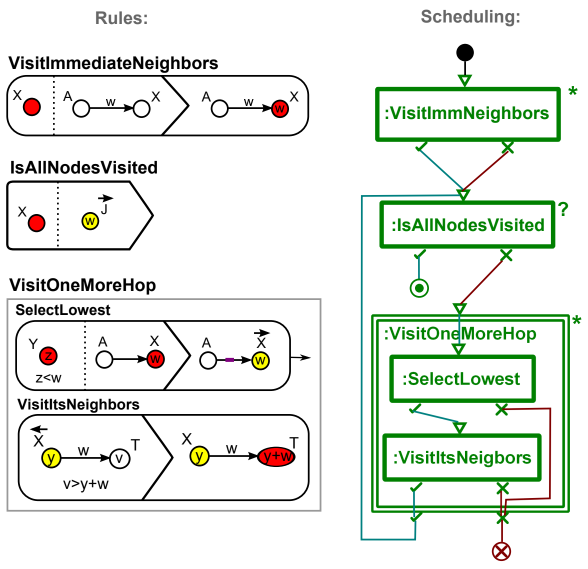


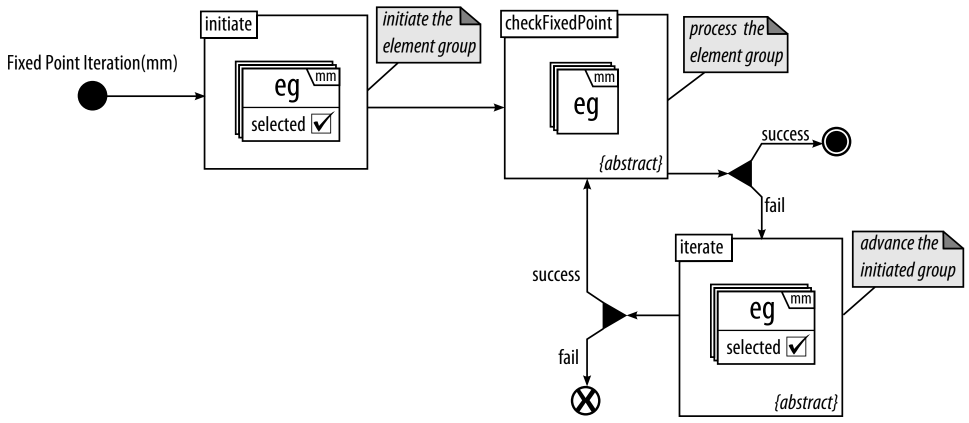
Figure 11: Rules for Dijkstra's Algorithm

6.3 Generalization of the Solution

The improved LCA, equivalent resistance, and Dijkstra's shortest path model transformation solutions look very alike. The structure is like a fixed-point iteration. In general there are three blocks. The first block initializes the input model with creation of some temporary elements and results in an intermediate form of the model (Initiate step). The initialization is optional (*e.g.*, Equivalent resistance problem in Section 6.2.1) but I have to include it in generalization. Then, a query verifies if a solution if found (Check step). Finally, if the query fails, the last block encodes one more step towards the solution (Advance step). The structure can also be seen as a `while not` loop in programming languages. I created the following pattern by using this generalization.

6.4 Fixed Point Iteration

- **Motivation:** 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. We previously identified this pattern in [44]. Asztalos *et al.* [45] also identified a similar structure named traverser model transformation analysis pattern.
- **Applicability:** 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.
- **Structure:** The structure is depicted in Figure 12. The fixed point iteration consists of rules that have abstract actions because processing at each iteration entirely depends on the application. Nevertheless, it enforces the following scheduling. The pattern starts by selecting a predetermined group of elements in the `initiate` rule and checks if the model has reached a fixed point (the condition is 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 `iterate` rule modifies the current model and the fixed point is checked again.



Fixed Point Iteration(mm)
initiate
eg
mm
selected
initiate the element group
checkFixedPoint
eg
mm
{abstract}
process the element group
success
fail
iterate
eg
mm
selected
{abstract}
advance the initiated group
success
fail

Figure 12: Structure of Fixed Point Iteration in Graphical Syntax

- **Examples:** In [44], we showed how to solve three problems with this pattern: computing the lowest common ancestor (LCA) of two nodes in a directed tree, which adds more elements to the input model; finding the equivalent resistance in an electrical circuit, which removes elements from the input model; and finding the shortest path using Dijkstra's algorithm, which only modifies attributes.
- **Implementation:** Figure 9 shows the implementation of the LCA in MoTif using the fixed point iteration pattern. The `initiate` rule is extended to create traceability links on the input nodes themselves with the `LinkToSelf` rules and with their parents with the `LinkToParent` rules. The `GetLCA` rule implements the `checkFixedPoint` rule and tries to find the LCA of the two nodes in the resulting model following traceability links. This rule does not have a RHS but it sets a pivot to the result for further processing. The `LinkToAncestor` rules implement the `iterate` rule by connecting the input nodes to their ancestors. The MoTif control structure reflects exactly the same scheduling with the design pattern.
- **Variations:** In some cases, the `initiate` rule can be omitted when, for instance, the `iterate` rule deletes selected elements such as in the computation of the equivalent resistance of an electrical circuit [44].

7 Additional Model Transformation Design Patterns

In this section, I show the additional model transformation design patterns (MTDP) by redefining the ones in existing studies or trying to identify new ones as in Section 6. A common practice while solving a problem is considered as a "design pattern" if we can apply it to different problems. This also implies the model transformation languages we use in the implementation can be different. Therefore, some useful practices within a single model transformation language should not be considered a design pattern, but on the contrary it can be a reusable idiom that can be supported with built-in structures. This list can be extended by other transformation developers, following the same style and using DelTa to represent the structure.

As mentioned in Section 6.3, each design pattern has some fields to describe. Inspired by the Gamma *et al.* [2] catalog templates, I use the following characteristics to describe a model transformation design pattern: *motivation* describes the need for and usefulness of the pattern, *applicability* outlines typical situations when the pattern can be applied, *structure* defines the pattern in DelTa and explains the pattern, *examples* illustrates practical cases where the patterns can be used, *implementation* provides a concrete implementation of the pattern in a MTL, and *variations* discusses some common variants of the pattern. In the structure characteristic, I use the graphical syntax of DelTa, but I also show the textual syntax in the first design pattern to give an idea how it looks like in action. For the example characteristic, I use a subset the UML class diagram metamodel with the concepts of class, attributes, and superclasses in most cases. For the implementation characteristic, I have implemented all design patterns in more than one languages such as MoTif, AGG, Henshin, Viatra2, GrGen.NET. Although I only show one implementation for each in this paper, the complete implementations can be found in [46]. This is how I validated the expressiveness, usability, and implementability of patterns defined in DelTa.

7.1 Entity Relation Mapping

- **Motivation:** Entity relation mapping (ER mapping) is one of the most commonly used transformation pattern in exogenous transformations encoding 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 [4] and later refined in [6].
- **Applicability:** The ER mapping is applicable when we want to translate elements from one metamodel into elements from another metamodel.
- **Structure:** The structure is depicted in Listing 2 in textual syntax and in Figure 13 in the graphical syntax. The pattern refers to two metamodels labeled `src` and `trgt`, corresponding to the source and target languages. It consists of a MTDP rule for mapping entities first and another for mapping relations. The `entityMapping` rule states that if an entity `e` from `src` is found, then an entity `f` must be created in `trgt` as well as a trace `t1` between them, if `t1` and `f` do not exist yet. The `relationMapping` rule states that if there is a relation `r1` between `e` and `f` in `src` and there is a trace `t1` between `e` and `g`, and a trace `t2` between `f` and `h`, then create a relation `r2` between `g` and `h` if it does not exist yet. Both rules should be applied recursively.

Listing 2: One-to-one Entity Relationship Mapping MTDP

```
mtdp OneToOneERMapping
    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
```

- **Examples:** A typical example of ER mapping 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:** I show the implementation of ER mapping in Henshin in Figure 14. The pattern states to apply the rules for entities before those for relations. Henshin provides a sequence structure with `SequentialUnit`. Henshin uses a compact notation for rules together with stereotypes on

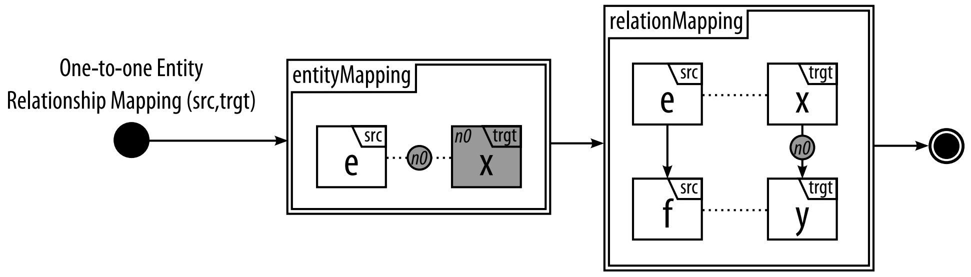


Figure 13: Structure of ER Mapping in Graphical Syntax

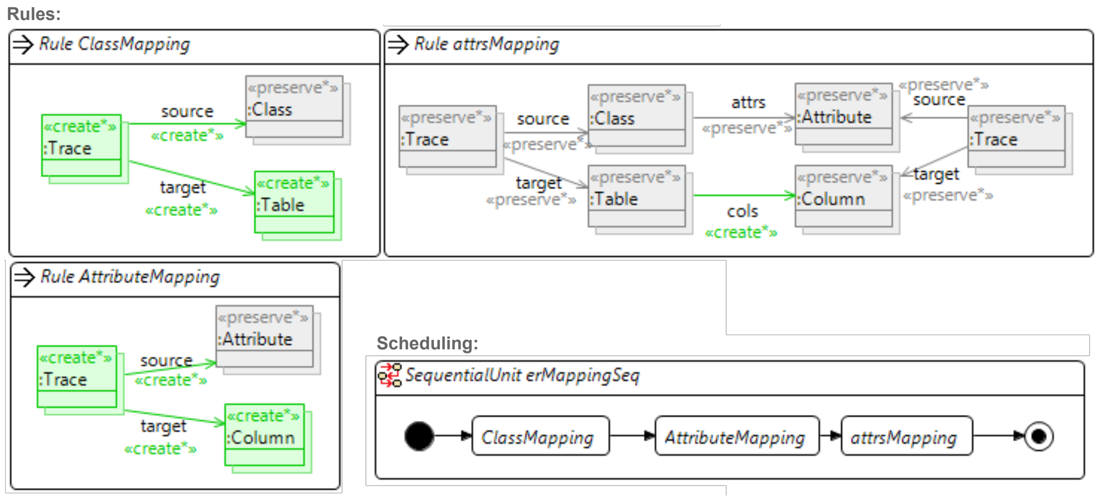


Figure 14: Rules of ER Mapping in Henshin

pattern elements. «preserve» is used for the elements found in the constraint of the MTDP rule and «create» is used to create elements found in the action of the MTDP rule. Here there are two rules corresponding to `entityMapping`: one for mapping classes to tables and one for mapping attributes to columns. In Henshin, traceability links must be modeled explicitly as a separate class connecting the source and target elements. I did not need to use NACs because Henshin provides a multi-node option that already prevents applying a rule more than once on the same match.

- **Variations:** Sometimes the entities in specific metamodels cannot be mapped one-to-one. It is possible to define one-to-many or many-to-many ER mappings pattern using element groups instead of entities (see [46]). Also, some implementations may require the creation of a trace between the two relations in the `relationMapping` rule.

7.2 Transitive Closure

- **Motivation:** Transitive closure is a pattern typically used for analyzing reachability related problems with an inplace transformation. It was proposed as a pattern in [3] and in [34]. It generates the intermediate paths between nodes that are not necessarily directly connected via traceability links.
- **Applicability:** The transitive closure pattern is applicable when the metamodels in the domain have a structure that can be considered as a directed tree.
- **Structure:** The structure is depicted in Figure 15. The pattern operates on single metamodel. First,

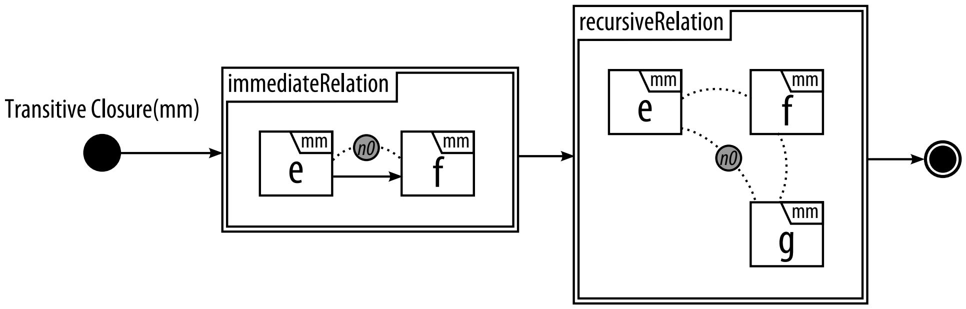


Figure 15: Structure of Transitive Closure in Graphical Syntax

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 `e-f` and `f-g` are connected with a trace, then `e` and `g` will also connected with a trace. It is also applied recursively to cover all nodes exhaustively.

- **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:** I have implemented the transitive closure in AGG. Figure 16 depicts the corre-

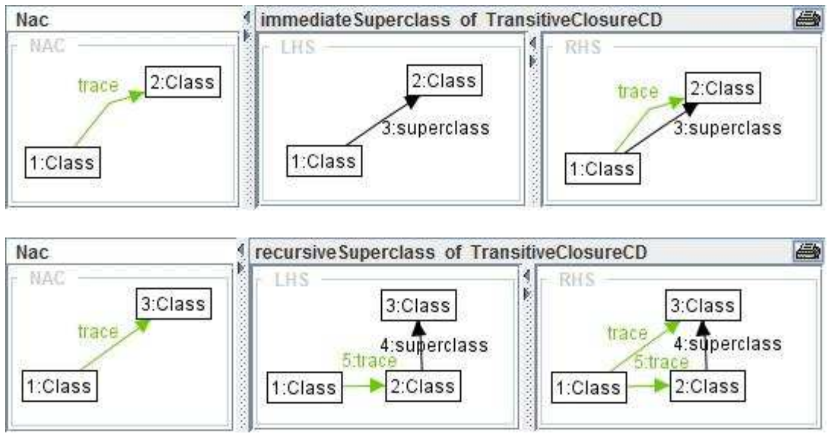


Figure 16: Transitive Closure rules in AGG

sponding rules. AGG rules consist of the traditional LHS, RHS, and NACs. The LHS and NACs represent the constraint of the MTDP rule and the RHS encodes the action. The `immediateSuperclass` rule creates a traceability link between a class and its superclass. The NAC prevents this traceability link from being created again. The `recursiveSuperclass` rule creates the remaining traceability links between a class and higher level superclasses. AGG lets the user 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, I 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.

- **Variations:** In some cases, a recursive `selfRelation` rule may be applied first, for example when computing the least common ancestor class of two classes, as in [44].

7.3 Visitor

- **Motivation:** The visitor pattern traverses all the nodes in a graph and processes each entity individually in a breadth-first fashion. This pattern is similar to the "leaf collector pattern" in [3] that is restricted to collecting the leaf nodes in a tree.
- **Applicability:** The visitor pattern can be applied to problems that consist of or can be mapped to any kind of graph structure where all nodes need to be processed individually.
- **Structure:** As depicted in Figure 17, the visitor pattern makes use of flags. The `markInitEntity`

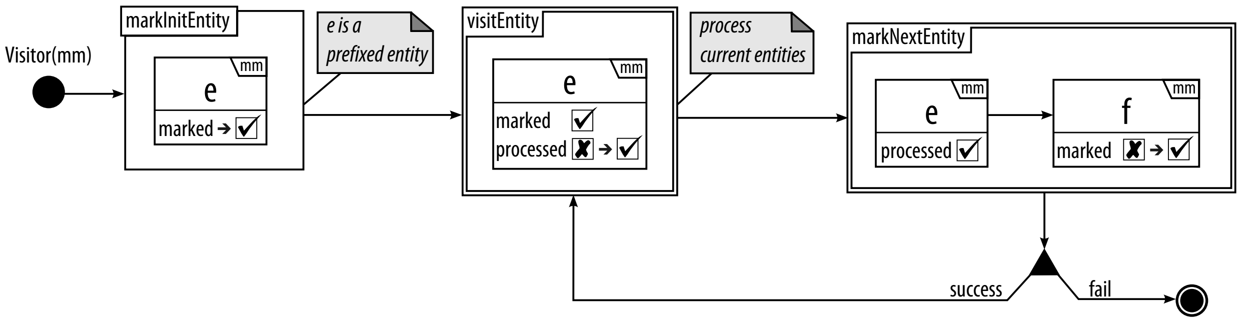
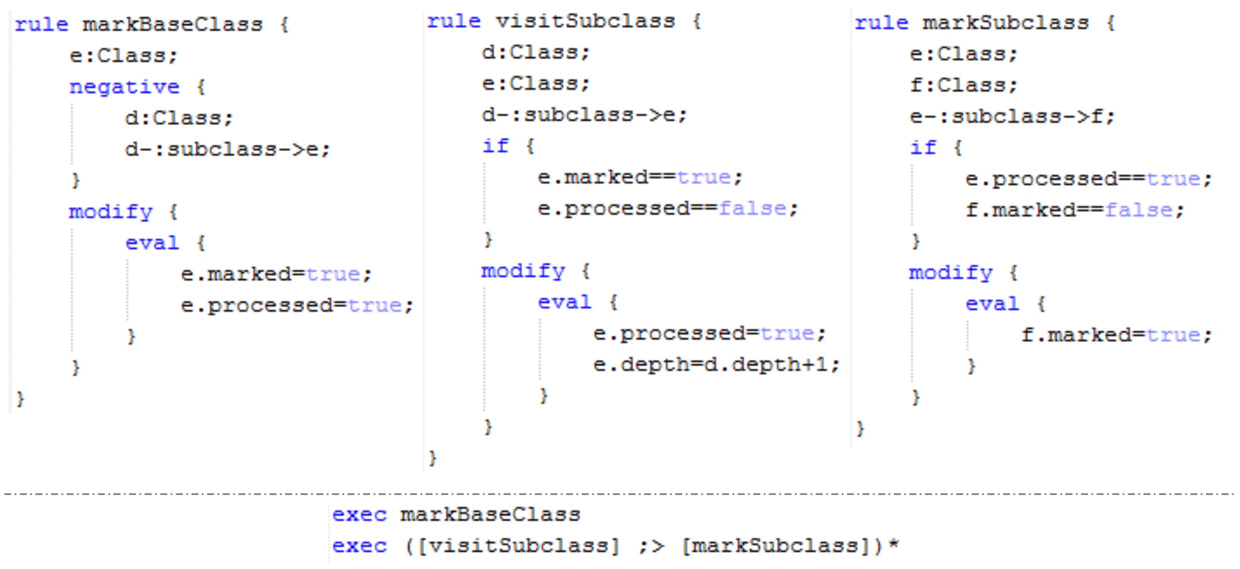


Figure 17: Structure of Visitor in Graphical Syntax

rule flags a predetermined initial entity as "marked". Note that in actual implementation, this rule maybe more complex. This rule is applied first and once. The next rule to be applied is the `visitEntity` rule. It visits the marked but unprocessed nodes by changing their processed flags to `true`. The actual processing of the node is left at the discretion of the implementation. Then, the `markNextEntity` rule marks the nodes that are adjacent to the processed nodes. Marking and processing are split into two steps to reflect the breadth-first traversal. The `markNextEntity` rule then initiates the loop to visit the remaining nodes. Visiting ends when `markNextEntity` is not applicable, *i.e.*, when all nodes are marked and have been processed.

- **Examples:** The visitor pattern helps to compute the depth level of each class in a class inheritance hierarchy, meaning its distance from the base class.
- **Implementation:** I have implemented visitor in GrGen.Net as depicted in Figure 18. 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. Since this class already has a depth level of 0, I 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 Figure 18. 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.



```
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 18: Visitor rules and scheduling in GrGen.Net

- **Variations:** It is possible to vary the traversal order, for example a depth-first strategy may be implemented. It is also possible to visit relations instead of entities. Another variation is to only have one recursive rule that processes all entities if the order in which they processed is irrelevant.

7.4 Execution by Translation

- **Motivation:** 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 developer to write an executor from scratch for the DSL and standardizes the execution in a way. 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.
- **Applicability:** The pattern is applicable when we want to execute a DSL and have another language to rely the simulation on.
- **Structure:** The structure of the pattern is depicted in Figure 19. The pattern refers to two meta-models; the `dsl`, which is the DSL we want to execute, and the `simLang`, which is the intermediate language we simulate instead of `dsl`. First, the `dsl` is mapped to the `simLang` by using the `OneToOneERMapping` design pattern described in [41]. This results in having each element in the `dsl` mapped to its corresponding equivalent in the `simLang`. Then, in the `init` 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 `simulateAndAnimate` transformation block is activated. In this block, 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. After this block, the `terminatingCondition` is checked again and the simulation goes on.
- **Examples:** In [38], Kühne *et al.* executes FSA by translating to PN. As they simulate the PN, they

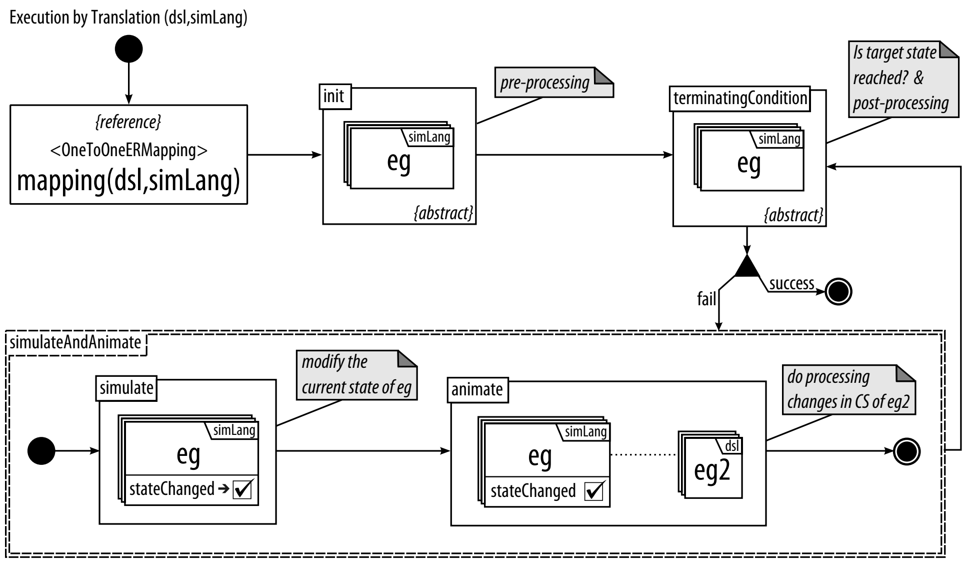


Figure 19: Execution by Translation Design Pattern Structure

animate the FSA accordingly. In [12], we have defined a translation from AD to PN, and simulated the PN to animate the AD. De Lara and Vangheluwe mapped production system DSL to PN and used PN for the dynamic behavior of production system in [47].

- **Implementation:** I implement PN to SC in MoTif [39]. In this example, the source language is executed and then the second language is animated, as described in the variants of the design pattern. The rules and scheduling are depicted in Figure 20. I only map the basic states and hyperedges in SC for simplicity, but the advanced transformation can be found in [48]. The `mapping` part maps the places to basic states and transitions to hyperedges with the `placeToBasicState` and the `transitionToHyperedge` rules. Then, the arcs of PN are mapped to links in SC with the `arcsToLinks` and the `arcsToLinksT2P` rules. After mapping, the `init` part is doing the same job as in the previous examples. The `setOneTokenToInitial` 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 rules are same as the previous examples, as they are regular PN simulation rules. In the `fullControlFlow` structure, one can realize that it looks similar to the structure of the "execution by translation" design pattern. This is because I inspire myself from existing model transformation languages while creating DelTa and the control flow of DelTa, which is the TURelation, consists of the primitives of MoTif scheduling structures.

- **Variations:** Usually, the simulation language, `simLang`, has fewer elements than the `dsl` language. In this case, the mapping part can be one-to-many, many-to-one or many-to-many entity relation mapping. One-to-many ER mapping is described in DelTa in [46]. Another variation is when the transformation simulates the first language and animates the second language accordingly. This

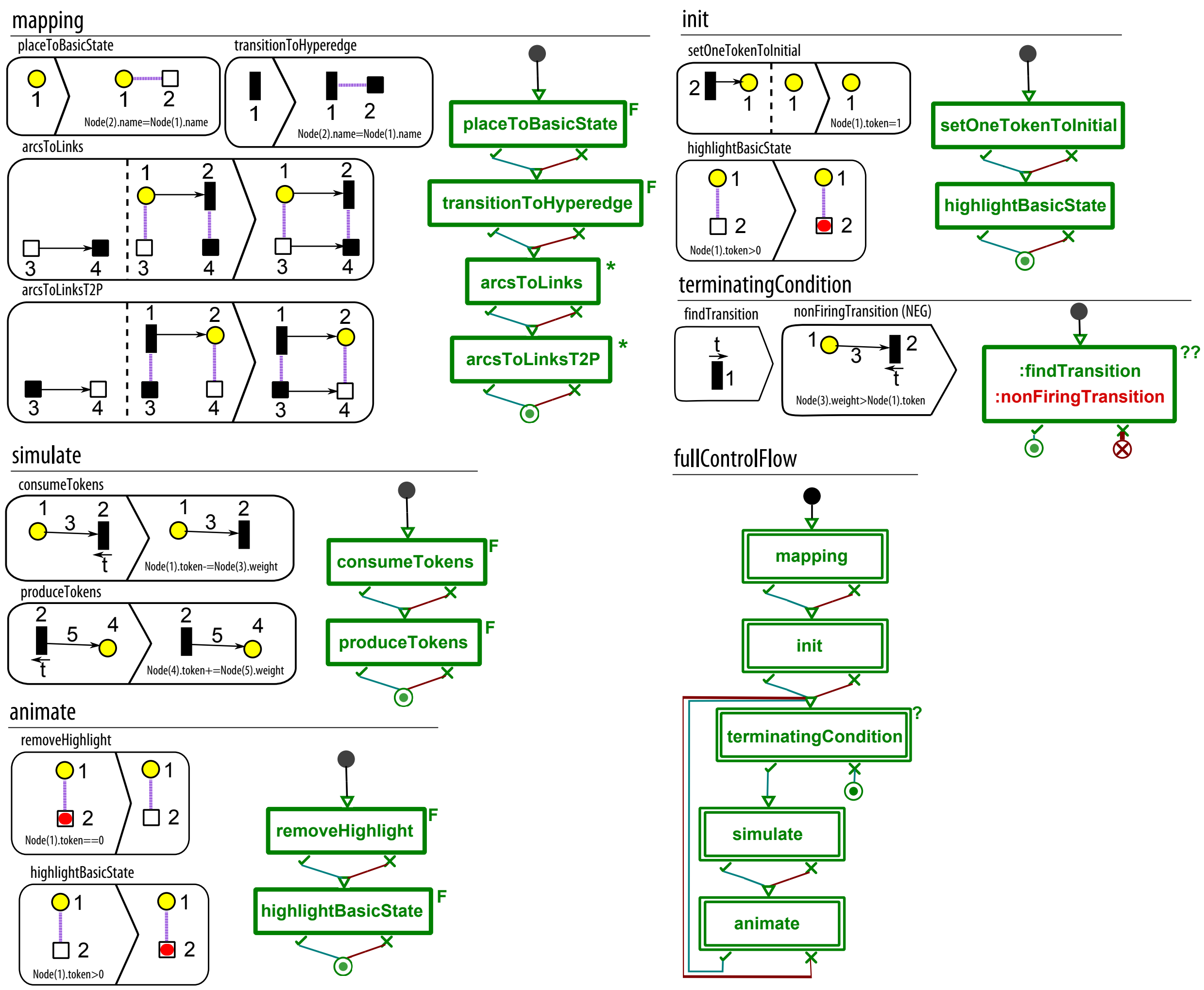


Figure 20: Petri Nets to statecharts in MoTif.

only inverts the two metamodels in the four rules of this design pattern.

8 Further Work & Schedule

In this section, I briefly explain what is the next phases of my research.

8.1 Identification of New Design Patterns and Intents Study

Identification of new design patterns is an important phase to complete the design pattern catalog. In the previous work, I have used two methods to identify a design pattern: 1) to solve different problems and try to come up with a common solution 2) to analyze existing studies.

For the latter, we are in the process of preparing a systematic literature review that will cover all model transformation related papers between 2003-2013, that have case studies, examples or demonstrations. The purpose of the work is to identify the intents of model transformations by using the real data. The study is a systematic literature review which will help to identify the intents of each model transformation and it will provide a systematic way of examining model transformations which may lead us to common practices and then design patterns. During that study, I will investigate a lot of model transformation papers which will help me to analyze how different problems are solved in different languages.

With more design patterns identified, I will also classify them like Gamma *et al.* did. They classified the object-oriented design patterns in three groups: creational, behavioral, and structural.

8.2 Uses of DelTa

The main purpose of DelTa is to provide an abstract language to express model transformation design patterns and help the transformation developers as a guideline. However, I also investigate how to automatically generate transformation using DelTa code. Model transformation languages are really diverse in terms of structure, therefore I have kept DelTa as abtract as possible. This brings the issue that DelTa cannot generate a transformation alone. For that purpose, I will check how to come up a Rule Diagram (RD) [1] like structure. In [1], authors create RDs for each language to generate the transformation. Therefore, I will also make use of specialized structures for each language and along with DelTa, I will use them to generate the transformation. The overall architecture is depicted in Figure 21. Each RD will

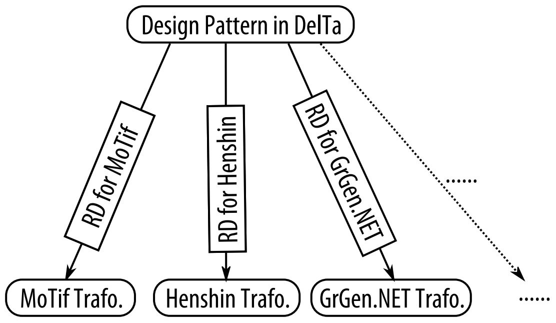


Figure 21: Transformation Generation in Different Languages

fill the necessary gap for each language to generate the transformations from DelTa.

8.3 Empirical Evaluation of DelTa

The empirical evaluation of DelTa consists of preparing an experiment for real transformation developers. For that reason, I choose Transformation Tool Contest (TTC), where real developers try to solve proposed problems in their own languages. For TTC, I will prepare a case study that will reveal all details of DelTa and let the developers implement them in their own choice of languages. This will give me a great insight about the usefullness and expresiveness of DelTa.

I conduct the experiment with the help of surveys and observational studies. The suryeys are to collect the experience of the developers with DelTa. The observational studies will be done in our university with the selected model transformation developers. They will be asked to develop a transformation from scratch. Later, they will be provided a design pattern that may help with the problem and some variables will be measured. One possible threat to the validity of this study is to find a good number of model transformation developers. I need two groups of model transformation developers; one for the control group and one for the experimental group. If I cannot find enough developers, I will provide necessary education on model transformation. I choose MoTif as the main transformation language for the first iterations of the user study. Then the case study in TTC will show how different languages are interacting with DelTa.

8.4 Detection of DelTa Design Patterns

As an initial start to detect design patterns in actual model transformations, I will use MoTif language. The effort consists of finding the design patterns instances in model transformations. Since DelTa is a language that has its own metamodel, detection of design patterns will be done by using an explicit model transformation. DelTa provides the rule and scheduling structures combined in its metamodel, whereas MoTif has the rule and scheduling structures in seperate metamodels, because a rule can be used regardless of the model transformation language. Therefore, the detection model transformation takes the DelTa design patterns and a model transformation designed in MoTif with the rules as inputs and tries to find a match. If the process succeeds, it will be expanded to other model transformation languages.

8.5 Schedule

Fall 2014

- Starting intents study to identify new design patterns
- Creating a modeling environment for DelTa in AToMPM
- Preparing of the empirical evaluation experiment and case study of DelTa
- Conducting the empirical evaluation on test subjects in our university
- Taking a numbered course

Spring 2015

- Investigating the uses of DelTa and how to generate transformations
- Conducting the empirical evaluation in TTC 2015
- Preparing the paper that consists of newly identified design patterns with the initial results from empirical evaluation and possible revisions to DelTa
- Preparing the "Software and Systems Modeling" journal paper which will have the DelTa in its final form, with all design patterns identified and the results of the full empirical evaluation
- Taking a special topics course
- Detecting the DelTa design patterns in actual model transformations
- Continuing on intents study

Summer 2015

- Write dissertation

Fall 2015

- Dissertation Defense & Graduation

9 Conclusion

In this proposal, I have listed a summary of what I am planning to do in my dissertation and what I have done. I have analyzed the model transformation development process and found out that in its current situation it is hard to develop model transformations. This is mostly because of the diversified nature of model transformation languages (MTL). Therefore, I have analyzed the structure of MTLs. At the same time, I have implemented the transformation engine (MoTif and T-Core) of AToMPM in Python. I have also created DSLs for these two languages. I have proposed a catalog of common practices that each transformation developer can adopt while creating their model transformation. I have named them model transformation design patterns. Currently, there are five model transformation design patterns. For each design pattern, I have created the necessary information to describe and implemented in five different model transformation languages. These languages are MoTif, Henshin, GrGen.NET, Viatra2, and AGG. As a result of analyzing the MTLs' structure, I have created the language DelTa to describe the structure of each design pattern. I have assigned a textual syntax to DelTa and generated a DelTa environment to edit design patterns using XText. DelTa is a concise and expressive language that I believe will play the role of UML as in object-oriented design patterns. The next steps include identifying more design patterns to reach to a more complete list like Gamma *et al.* [2] did for object-oriented community. I will also support DelTa with empirical experiments.

References

[1] Guerra, E., de Lara, J., Kolovos, D., Paige, R., and dos Santos, O. (2013) Engineering model transformations with transML. *Software and Systems Modeling*, **12**, 555–577.

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Boston, MA, USA.

[3] Agrawal, A. (2005) Reusable Idioms and Patterns in Graph Transformation Languages. *International Workshop on Graph-Based Tools*, ENTCS, **127**, pp. 181–192. Elsevier.

[4] Iacob, M.-E., Steen, M. W. A., and Heerink, L. (2008) Reusable Model Transformation Patterns. *EDOC Workshops*, September, pp. 1–10. IEEE Computer Society.

[5] Bézivin, J., Jouault, F., and Paliès, J. (2005) Towards model transformation design patterns. *Proceedings of the First European Workshop on Model Transformations (EWMT 2005)*.

[6] Kevin Lano and Shekoufeh Kolahdouz Rahimi (2013) Constraint-based specification of model transformations. *Journal of Systems and Software*, **86**, 412–436.

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

[8] Kleppe, A. G., Warmer, J., and Bast, W. (2003) *MDA Explained. The Model Driven Architecture: Practice And Promise*. Addison-Wesley.

[9] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008) ATL: A model transformation tool. *Science of Computer Programming*, **72**, 31–39.

[10] Amrani, M., Dingel, J., Lambers, L., Lucio, L., Salay, R., Selim, G., Syriani, E., and Wimmer, M. (2012) Towards a Model Transformation Intent Catalog. *MoDELS workshop on Analysis of model Transformation*. IEEE.

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

[12] Syriani, E. and Ergin, H. (2012) Operational Semantics of UML Activity Diagram: An Application in Project Management. *RE 2012 Workshops, IEEE, Chicago*.

[13] Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. (2010) Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. *MODELS 2010*, LNCS, **6394**, pp. 121–135. Springer.

[14] Agrawal, A., Karsai, G., Kalmar, Z., Neema, S., Shi, F., and Vizhanyo, A. (2006) The Design of a Language for Model Transformations. *Journal on Software and Systems Modeling*, **5**, 261–288.

[15] Klein, T., Nickel, U., Niere, J., and Zündorf, A. (1999) From UML to Java And Back Again. Technical Report tr-ri-00-216. University of Paderborn, Paderborn.

[16] Varró, D. and Balogh, A. (2007) The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, **68**, 214–234.

[17] Taentzer, G. (2004) AGG: A graph transformation environment for modeling and validation of software. *AGTIVE*, pp. 446–453. Springer.

[18] Jouault, F. and Kurtev, I. (2007) On the interoperability of model-to-model transformation languages. *Science of Computer Programming, Special Issue on Model Transformation*, **68**, 114–137.

[19] Syriani, E., Vangheluwe, H., and LaShomb, B. (2013) T-Core: a framework for custom-built model transformation engines. *Software & Systems Modeling*, **13**, 1–29.

[20] Syriani, E., Gray, J., and Vangheluwe, H. (2012) Modeling a Model Transformation Language. *Domain Engineering: Product Lines, Conceptual Models, and Languages*. Springer.

[21] Syriani, E. and Vangheluwe, H. (2010) De-/Re-constructing Model Transformation Languages. *EASST*, **29**.

[22] Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., and Ergin, H. (2013) Atompm: A web-based modeling environment. *MODELS'13: Invited Talks, Demos, Posters, and ACM SRC. CEUR-WS.org*.

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

[24] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996) *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA.

[25] Mowbray, T. J. and Malveau, R. C. (1997) *CORBA Design Patterns*. Wiley.

[26] Douglass, B. P. (2002) *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[27] Buschmann, F., Henney, K., and Schmidt, D. C. (2007) *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Wiley.

[28] Wahba, S. K., Hallstrom, J. O., and Soundarajan, N. (2010) Initiating a design pattern catalog for embedded network systems. *Proceedings of the Tenth ACM International Conference on Embedded Software*, New York, NY, USA EMSOFT '10, pp. 249–258. ACM.

[29] Cho, H. and Gray, J. (2011) Design patterns for metamodels. *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, &#38; VMIL'11*, New York, NY, USA SPLASH '11 Workshops, pp. 25–32. ACM.

[30] Krasner, G., Pope, S., et al. (1988) A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, **1**, 26–49.

[31] Hasheminejad, S. M. H. and Jalili, S. (2012) Design patterns selection: An automatic two-phase method. *Journal of Systems and Software*, **85**, 408–424.

[32] Correa, A., Werner, C., and Zaverucha, G. (2000) Object oriented design expertise reuse: An approach based on heuristics, design patterns and anti-patterns. In Frakes, W. (ed.), *Software Reuse: Advances in Software Reusability*, Lecture Notes in Computer Science, **1844**, pp. 336–352. Springer Berlin Heidelberg.

[33] Blomqvist, E. (2008) Pattern ranking for semi-automatic ontology construction. *Proceedings of the 2008 ACM Symposium on Applied Computing*, New York, NY, USA SAC '08, pp. 2248–2255. ACM.

[34] Levendovszky, T., Lengyel, L., and Mészáros, T. (2009) Supporting domain-specific model patterns with metamodeling. *Software & Systems Modeling*, **8**, 501–520.

[35] Dong, J., Zhao, Y., and Peng, T. (2009) A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering*, **19**, 823–855.

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

[37] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. (2006) *Fundamentals of Algebraic Graph Transformation* EATCS. Springer-Verlag.

[38] Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., and Wimmer, M. (2010) Explicit Transformation Modeling. *Models in Software Engineering*, Lecture Notes in Computer Science, **6002**, pp. 240–255. Springer Berlin Heidelberg.

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

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

[41] Ergin, H. and Syriani, E. (2014) Towards a Language for Graph-Based Model Transformation Design Patterns. *Theory and Practice of Model Transformation, LNCS*, York, U.K., July, pp. 91–105. Springer.

[42] Geiß, R. and Kroll, M. (2008) GrGen. net: A fast, expressive, and general purpose graph rewrite tool. *Applications of Graph Transformations with Industrial Relevance*, pp. 568–569. Springer.

[43] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1973) On finding lowest common ancestors in trees. *Proceedings of the fifth annual ACM symposium on Theory of computing*, New York, NY, USA STOC '73, pp. 253–265. ACM.

[44] Ergin, H. and Syriani, E. (2013) Identification and Application of a Model Transformation Design Pattern. *ACM Southeast Conference*, Savannah GA, apr ACMSE'13. ACM.

[45] Asztalos, M., Madari, I., and Lengyel, L. (2010) Towards formal analysis of multi-paradigm model transformations. *SIMULATION*, **86**, 429–452.

[46] Ergin, H. and Syriani, E. (2014) Implementations of Model Transformation Design Patterns Expressed in DelTa. Technical Report SERG-2014-01. University of Alabama, Department of Computer Science.

[47] de Lara, J. and Vangheluwe, H. (2010) Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, **22**, 297–326.

[48] Ergin, H. and Syriani, E. (2013) AToMPM Solution for the Petri Net to Statecharts Case Study. *Seventh Transformation Tool Contest*, jul.

Appendix A List of Papers

A.1 Published

- Eugene Syriani and Huseyin Ergin. *Operational Semantics of UML Activity Diagram: An Application in Project Management*. Requirement Engineering Conference 2012 Workshops, IEEE, Chicago, IL (September 2012)
- Huseyin Ergin and Eugene Syriani. *Identification and Application of a Model Transformation Design Pattern*. ACM Southeast Conference 2013, Savannah, GA (April 2013)
- Huseyin Ergin. *Model Transformation Design Patterns*. MODELS Conference 2013 Doctoral Symposium, Miami, FL (October 2013)
- Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon van Mierlo, and Huseyin Ergin. *AToMPM: A Web-based Modeling Environment*. MODELS Conference 2013 Demonstrations, Miami, FL (October 2013)

Technical Report

- Huseyin Ergin and Eugene Syriani. *Implementations of Model Transformation Design Patterns Expressed in DelTa*. Department of Computer Science, University of Alabama, SERG-2014-01 (February 2014)

A.2 Accepted & To Be Presented

- Huseyin Ergin and Eugene Syriani. *Towards A Language To Express Design Patterns for Graph-Based Model Transformation*. International Conference on Model Transformation 2014, York, UK (July 2014)
- Huseyin Ergin and Eugene Syriani. *AToMPM Solution for the IMDB Case Study*. Transformation Tool Contest 2014, York, UK (July 2014)

A.3 Submitted

- Huseyin Ergin and Eugene Syriani. *Reuse of Model Transformation Design Patterns*. 8th System Analysis and Modelling Conference 2014, Valencia, Spain (September 2014)

A.4 In Preparation & Planning

- Huseyin Ergin and Eugene Syriani. *DelTa: A Language for Model Transformation Design Patterns*. Journal of Software and Systems Modeling (2015)
- Huseyin Ergin and Eugene Syriani. *Model Transformation Design Patterns in Action: Experiences with DelTa*. Automated Software Engineering (2015)
- Huseyin Ergin and Eugene Syriani. *The Experiences on How To Generate Transformations Using DelTa*. International Conference on Model Transformation (2015)
- Eugene Syriani, Jeffrey Carver, Huseyin Ergin, and Ahmet AlZubidy. *Model Transformation Intents: A Systematic Literature Review*. Empirical Software Engineering Conference (2015)
- Huseyin Ergin and Eugene Syriani. *DelTa Case Study*. Transformation Tool Contest (2015)