Towards a Language for Graph-Based Model Transformation Design Patterns

Hüseyin Ergin and Eugene Syriani

University of Alabama, U.S.A.
hergin@crimson.ua.edu,esyriani@cs.ua.edu

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 design, we believe that "good design" of model transformation can benefit tremendously from model transformation design patterns. Hence, in this paper, we present DelTa, a language for expressing design patterns for model transformations. DelTa is more abstract than and independent from any existing model transformation language, yet it is expressive enough to define design patterns as guidelines transformation developers can follow. To validate the language, we have redefined four known model transformation design patterns in DelTa and demonstrated how such abstract transformation guidelines can be implemented in five different model transformation languages.

1 Introduction

Model-driven engineering 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. One of the most essential contribution to software design was the GoF catalog of object-oriented design patterns [2]. Similarly, we believe that the design of model transformations can tremendously benefit from model transformation design patterns. Although very few design patterns have been proposed in the past ([3,4,5,6,7]), they were each expressed in a specific model transformation language (MTL) and hence hardly re-usable in any other.

As stated in [8], a design pattern language must be independent from any MTL in which patterns are implemented. Furthermore, it must be fit to define *patterns* rather than *transformations*. For example, GoF design patterns are described in UML class diagram which is independent from the object-oriented programming language used for the implementation of software. A design pattern language must also be understandable and implementable by a transformation developer. Additionally, it must allow one to verify if a transformation correctly implements a pattern. To satisfy the language independence and implementability requirements, this paper proposes DelTa, a domain-specific language to describe design patterns for model transformations. Furthermore,

DelTa is expressive enough to define design patterns as guidelines transformation developers can follow. Note that DelTa currently focuses on graph-based model transformation only.

In Section 2, we present the syntax and informal semantics of DelTa. To validate the language in Section 3, we redefine four known model transformation design patterns using DelTa and demonstrate how design patterns expressed in DelTa can be implemented in existing graph-based MTLs. In Section 4, we discuss related work. We finally discuss limitations of our approach and conclude in Section 5.

2 Design Pattern Language for Graph-based Model Transformation

DelTa is a neutral language, independent from any MTL. It is designed to define design patterns for model transformations, hence it is not a language to define model transformations. We could have used an existing MTL as a notation for DelTa, however our need is a notation that expresses how elements within a rule are related and how rules are related with each other. In this respect, DelTa offers 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 to define design patterns for object-oriented languages. In the following, we describe the abstract syntax, concrete syntax, and informal semantics of DelTa.

2.1 Abstract Syntax

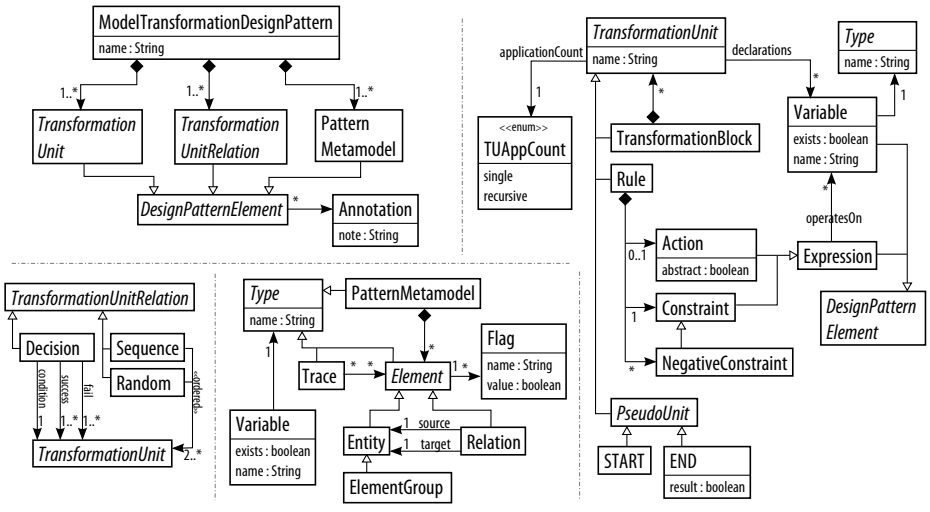

Fig. 1. DelTa Metamodel

As depicted in Fig. 1, a model transformation design pattern (MTDP) consists of three kinds of components: transformation units (TU), pattern elements and transformation unit relations (TUR). This is consistent with the structure of common MTLs [9]. TUs represent the concept of rule in graph-based model transformations [10]. 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 MTPD, 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 [11]. 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 [12] and GReAT [13], and parameter passing in Viatra2 [14]. 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, we 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 [15], 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 [16] 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 [15]. For reuse purposes, rules in DelTa can be grouped into transformation blocks, similarly to a Block in GReAT.

As surveyed in [12,17], in any MTL, rules are subject to a scheduling policy, whether it is implicit or explicit. For example, AGG [18] uses layers, MoTif and VMTS [19] use a control flow language, and GReAT defines causality relations between rules. As shown in [20], 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 2.3. 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.

2.2 Concrete Syntax

Listing 1.1. EBNF Grammar of DelTa in XText

```
MTDP:   'mtdp' NAME
          'metamodels:' NAME (',' NAME)* ANNOTATION?
           (('tblock' NAME '*'? ANNOTATION?)?
              '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') ':' ;
```

We opted for a textual concrete syntax for DelTa. Listing 1.1 shows the EBNF grammar implemented in Xtext. 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. 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 '$\sim$' 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 '#'. Listings 1.2– 1.5 show concrete examples of MTDPs using this notation.

2.3 Informal Semantics

The semantics of MTDP rules is borrowed from graph transformation rules [10], 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 [21]. 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.

3 Model Transformation Design Patterns

In this section, we illustrate how to use DelTa pragmatically by redefining four existing design patterns for MT. Inspired by the GoF catalog templates, we describe a MTDP using the following characteristics: *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. For the example characteristic, we use a subset the UML class diagram metamodel with the concepts of class, attributes, and superclasses. For the implementation characteristic, we have implemented all design patterns in five languages: MoTif, AGG, Henshin, Viatra2, GrGen.NET. Although we only show one implementation for each in this paper, the complete implementations can be found in [22]. This is how we validated the expressiveness, usability, and implementability of patterns defined in DelTa.

3.1 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 [6] and later refined in [23].
- **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 1.2. 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 1.2. 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:** We show the implementation of ER mapping in Henshin in

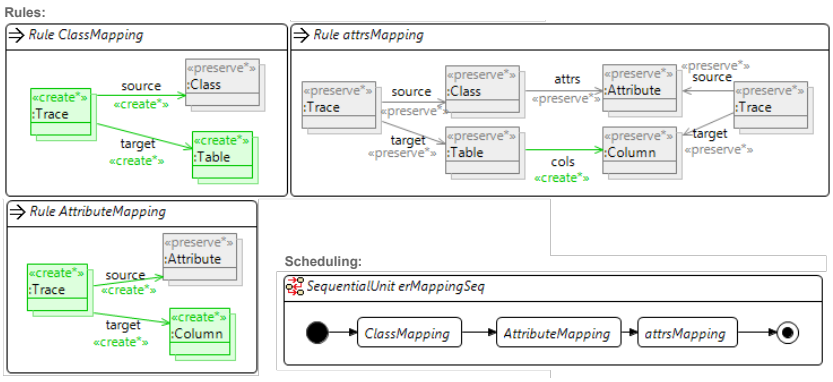

Fig. 2. Rules of ER Mapping in Henshin

Fig. 2. 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 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. We 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 [22]). Also, some implementations may require the creation of a trace between the two relations in the relationMapping rule.

3.2 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 [24]. 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:**

Listing 1.3. Transitive Closure MTDP

```
mtdp TransitiveClosure
    metamodels: mm
    rule immediateRelation*
        Entity mm.e, mm.f
        Relation r1(mm.e, mm.f)
        Trace t1(mm.e, mm.f)
        constraint: mm.e, mm.f, r1, ~t1
        action: t1
    rule recursiveRelation*
        Entity mm.a, mm.b, mm.c
        Trace t1(mm.a, mm.b), t2(mm.b, mm.c), t3(mm.a, mm.c)
        constraint: mm.a, mm.b, mm.c, t1, t2, ~t3
        action: t3
    Sequence: START, immediateRelation, recursiveRelation, END
```

The structure is depicted in Listing 1.3. The pattern operates on single metamodel. First, the immediateRelation rule creates a trace element between entities connected with a relation. It is applied recursively to cover all relations. Then, the recursiveRelation rule creates trace elements between the node indirectly connected. That is if entities `a-b` and `b-c` are connected with a trace, then `a` and `c` 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:** We have implemented the transitive closure in AGG. Fig. 3 de-

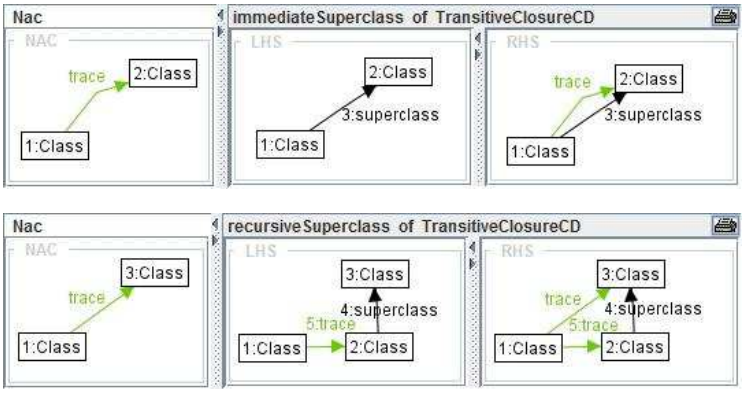

Fig. 3. Transitive Closure rules in AGG

picts the corresponding rules. AGG rules consist of the traditional LHS, RHS, and NACs. The LHS and NACs represent the constraint of the MTDP rule and the RHS encodes the action. The immediateSuperclass rule creates a traceability link between a class and its superclass. The NAC prevents this traceability link from being created again. The recursiveSuperclass rule creates the remaining traceability links

between a class and higher level superclasses. AGG lets the 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, we set the layer of immediateSuperclass to 0 and recursiveSuperclass to 1 as the design pattern structure stated these rules to be applied in a sequence.

– **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 [5].

3.3 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:**

Listing 1.4. Visitor MTDP

```
mtdp Visitor
    metamodels: mm
    rule markInitEntity
        Entity mm.e
        constraint: mm.e # e is a predetermined entity #
        action: mm.e[marked=true]
    rule visitEntity*
        Entity mm.e
        constraint: mm.e[marked=true,processed=false]
        action: mm.e[processed=true] # Process current entities #
    rule markNextEntity*
        Entity mm.e, mm.f
        Relation r1(mm.e, mm.f)
        constraint: mm.e[processed=true], mm.f[marked=false], r1
        action: mm.f[marked=true]
    Sequence: START, markInitEntity, visitEntity, markNextEntity
    markNextEntity ? visitEntity : END
```

As depicted in Listing 1.4, the visitor pattern makes use of flags. The markInitEntity 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.

Fig. 4. Visitor rules and scheduling in GrGen.NET

– **Implementation:** We have implemented visitor in GrGen.NET as depicted in Fig. 4. 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, we flag it as processed to prevent the visitSubclass rule from increasing its depth. This is a clear example of the minimality of a MTDP rule, where the implementation extends the rule according to the application. The visitSubclass rule processes the marked elements. Here, processing consists of increasing the depth of the subclass by one more than the depth of the superclass. The markSubclass rule marks subclasses of already marked classes. The scheduling of these GrGen.NET rules is depicted in the bottom of Fig. 4. 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.
– **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.

3.4 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 [5]. Asztalos *et al.* [25]

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:**

Listing 1.5. Fixed Point Iteration MTDP

```
mtdp FixedPointIteration
    metamodels: mm
    rule initiate
        ElementGroup mm.eg1
        constraint: mm.eg1
        action: mm.eg1[selected=true] # Initiate the element group #
    rule checkFixedPoint
        ElementGroup mm.eg1
        constraint: mm.eg1
        abstract action: # Process the element group #
    rule iterate
        ElementGroup mm.eg1
        constraint: mm.eg1[selected=true]
        abstract action: # Advance the initiated group #
    Sequence: START, initiate, checkFixedPoint
    checkFixedPoint ? END[result=true] : iterate
    iterate ? checkFixedPoint : END[result=false]
```

The structure is depicted in Listing 1.5. 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.

– **Examples:** In [5], 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:** Fig. 5 shows the implementation of the LCA from [5] in MoTif using the fixed point iteration pattern. The initiate rule is extended to create traceability links on the input nodes themselves with the LinkToSelf rules and with their parents with the LinkToParent rules. The GetLCA rule implements the checkFixedPoint rule and tries to find the LCA of the two nodes in the resulting model following traceability links. This rule does not have a RHS but it sets a pivot to the result for further processing. The LinkToAncestor rules implement the iterate rule by connecting the input nodes to their ancestors. The MoTif control structure reflects exactly the same scheduling of Listing 1.5.
– **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 [5].

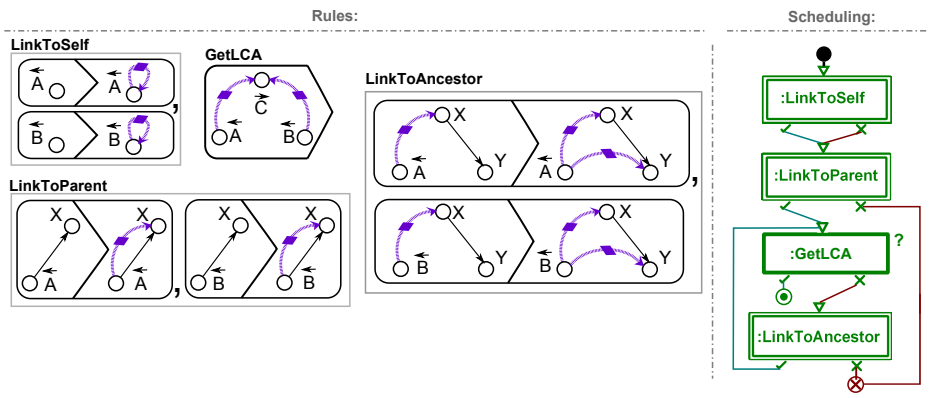

Fig. 5. Rules and Scheduling in MoTif

4 Related Work

The first work that proposed design patterns for model transformation was by Agrawal *et al.* [3]. They defined the *transitive closure* pattern which is similar to what we showed in Section 3.2, except that we create traceability links whereas they reuse the same association type from the input metamodel. 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 3.3 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.* [6] defined five other design patterns for outplace transformations. Similar to the ER mapping pattern in Section 3.1, 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.* [7] 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.

The first issue with these three previous works is that all the design patterns are defined using GReAT, QVT-R, and ATL 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 application domains. On the contrary, DelTa is independent from any MTL and defines the patterns using abstracted elements independent from the input and output metamodels.

Lano *et al.* [23] 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. The fixed point iteration pattern in Section 3.4 can be used in one of the creation phases here. 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, which is similar to what the TURs of DelTa offer.

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. Like DelTa, RD is defined at a level of abstraction that is independent from existing MTLs. But its purpose is to generate a transformation rather than to define design patterns. Therefore, there are some similarities and differences between RD and DelTa. 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 which is similar to the decision TUR in DelTa, or non-deterministically choosing to apply one rule which is similar to the random TUR. They also allow rules to explicitly invoke the application of other rules. RD is inspired from QVT-R and ETL and is therefore more easily implementable in these language, whereas DelTa currently focuses on graph-based MTLs.

Levendovszky *et al.* [24] 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, which is similar to the transitive closure pattern in Section 3.2.

5 Conclusion

In this paper, we proposed DelTa as candidate for a design pattern language for model transformations. DelTa is a language for describing patterns, rather than transformations. It is independent from any MTL yet directly implementable in most graph-based MTLs. To validate the language, we described four known design patterns for model transformation and implemented them in five different languages (the complete implementations can be found in [22]).

As stated in Section 1, a design pattern language must also be understandable and suited to verify correct implementations. For the former, we plan to empirically evaluate DelTa by running user studies. The verifiability requirement remains to be investigated. A formal specification language such as in [23] can then be used, but at the price of the understandability and ease of implementability. Furthermore, identifying additional design patterns will help us evolve the DelTa language and further validate its expressiveness.

When implementing the design patterns, we realized that some patterns are easier to implement in some languages than in others due the constructs they offer for transformation units and for scheduling. In particular, when implementing a pattern that involves more complex scheduling (such as the fixed point iteration) in MTLs with very limited scheduling policies (such as AGG), several tricks need to be used, such as modifying the metamodel or making use of temporary elements or attributes. The lack of a standard paradigm for model transformations is the main source of this difficulty that the model transformation community has to agree on. We plan to extend DelTa to cover non-graph-based MTLs, such as QVT-OM and ATL, and possibly bi-directional MTLs, such as QVT-R and triple graph grammars.

References

1. Guerra, E., de Lara, J., Kolovos, D., Paige, R., dos Santos, O.: Engineering model transformations with transML. Software and Systems Modeling **12** (2013) 555–577
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, Boston, MA, USA (1995)
3. Agrawal, A.: Reusable Idioms and Patterns in Graph Transformation Languages. In: International Workshop on Graph-Based Tools. Volume 127 of ENTCS., Elsevier (2005) 181–192
4. Bézivin, J., Rumpe, B., Tratt, L.: Model Transformation in Practice Workshop Announcement (2005)
5. Ergin, H., Syriani, E.: Identification and Application of a Model Transformation Design Pattern. In: ACM Southeast Conference. ACMSE'13, Savannah GA, ACM (apr 2013)
6. Iacob, M.E., Steen, M.W.A., Heerink, L.: Reusable Model Transformation Patterns. In: EDOC Workshops, IEEE Computer Society (September 2008) 1–10
7. Bézivin, J., Jouault, F., Paliès, J.: Towards model transformation design patterns. In: Proceedings of the First European Workshop on Model Transformations (EWMT 2005). (2005)

8. Syriani, E., Gray, J.: Challenges for Addressing Quality Factors in Model Transformation. In: Software Testing, Verification and Validation. ICST'12, IEEE (apr 2012) 929–937
9. Syriani, E., Gray, J., Vangheluwe, H.: Modeling a Model Transformation Language. In: Domain Engineering: Product Lines, Conceptual Models, and Languages. Springer (2012)
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS. Springer-Verlag (2006)
11. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit Transformation Modeling. In: MODELS 2009 Workshops. Volume 6002 of LNCS., Springer (2010) 240–255
12. Syriani, E., Vangheluwe, H.: A Modular Timed Model Transformation Language. Journal on Software and Systems Modeling **12**(2) (jun 2011) 387–414
13. Agrawal, A., Karsai, G., Shi, F.: Graph transformations on domain-specific models. Journal on Software and Systems Modeling (2003)
14. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming **68**(3) (2007) 214–234
15. Syriani, E., Vangheluwe, H., LaShomb, B.: T-Core: a framework for custom-built model transformation engines. Software & Systems Modeling (2013) 1–29
16. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: MODELS 2010. Volume 6394 of LNCS., Springer (2010) 121–135
17. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal **45**(3) (jul 2006) 621–645
18. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: AGTIVE. Springer (2004) 446–453
19. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model Transformation with a Visual Control Flow Language. International Journal of Computer Science **1**(1) (2006) 45–53
20. Syriani, E., Vangheluwe, H.: De-/Re-constructing Model Transformation Languages. EASST **29** (mar 2010)
21. Geiß, R., Kroll, M.: GrGen. net: A fast, expressive, and general purpose graph rewrite tool. In: Applications of Graph Transformations with Industrial Relevance. Springer (2008) 568–569
22. Ergin, H., Syriani, E.: Implementations of Model Transformation Design Patterns Expressed in DelTa. `http://software.eng.ua.edu/reports/SERG-2014-01.pdf` SERG-2014-01, University of Alabama, Department of Computer Science (feb 2014)
23. Kevin Lano, Shekoufeh Kolahdouz Rahimi: Constraint-based specification of model transformations. Journal of Systems and Software **86**(2) (2013) 412–436
24. Levendovszky, T., Lengyel, L., Mészáros, T.: Supporting domain-specific model patterns with metamodeling. Software & Systems Modeling **8**(4) (2009) 501–520
25. Asztalos, M., Madari, I., Lengyel, L.: Towards formal analysis of multi-paradigm model transformations. SIMULATION **86**(7) (2010) 429–452