Implementations of Model Transformation Design Patterns Expressed in DelTa

Hüseyin Ergin and Eugene Syriani

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

Abstract. DelTa is a neutral language, independent from any model transformation language (MTL). It is expressive enough to define design patterns as guidelines transformation developers can follow. In this technical report, we revisit what DelTa is, redefine four known model transformation design patterns in DelTa and demonstrate the implementation in five MTLs.

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.

In this technical report, we revisit the syntax and semantics of 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. In Section 3, we redefine four known model transformation design patterns using DelTa. We finally conclude in Section 4.

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. In this respect, it 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

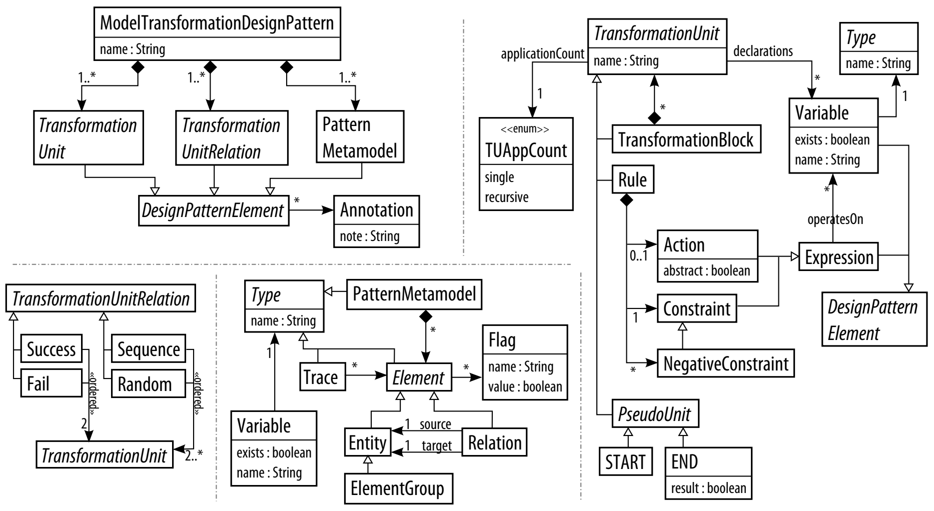


Fig. 1. DelTa Metamodel

As depicted in Fig. 1, a model transformation design pattern (MTDP) consists of three components: transformation units (TU), pattern elements and transformation unit relations (TUR). This is consistent with the structure of common MTLs [8]. TUs represent the concept of rule in graph-based model transformations [9]. 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 [10].

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 [11] and GReAT [12], and parameter passing in Viatra2 [13]. 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, traces 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 traces in the implementation.

As surveyed in [14], 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 [15] 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 [14]. For reuse purposes, rules in DelTa can be grouped into transformation blocks, similarly to a Block in GReAT.

As surveyed in [11,16], in any MTL, rules are subject to a scheduling policy, whether it is implicit or explicit. For example, AGG [17] uses layers, MoTif and VMTS [18] use a control flow language, and GReAT defines causality relations between rules. As shown in [19], 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 four TURs of DelTa: Sequence, Random, Success, and Fail that are explained in Section 2.3. The former two act on at least two TUs and the latter two on exactly two 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 (',' '~'? ELEMENTNAME)* ANNOTATION? ;
NegativeConstraint: 'negative constraint:' ELEMENTNAME (',' ELEMENTNAME)*
                     ANNOTATION? ;
Action: ('abstract action' | 'action:' ('~'? ELEMENTNAME (',' '~'? ELEMENTNAME)*))
              ANNOTATION? ;
TURelation: TURTYPE ('START' | (NAME ('[' NAME '=' ('true' | 'false')']')?))
              (',' ('END' | NAME) ('[' 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' | 'Success' | 'Fail' | '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. Annotations are enclosed within '$\#$'. Listings 1.2– 1.6 show concrete examples of MTDPs using this notation.

2.3 Informal Semantics

The semantics of MTDP rules is borrowed from graph transformation rules [9], 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 [20]. 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.

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 Success and Fail relations define causal relationships between two TUs in case the first is applicable or not respectively. For example `Success:A,B` means that if `A` is applicable then `B` should be applied after. Note that the latter two TURs can be used to define loop structures. For example, `Success: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`. The scheduling `Success:T,R` 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. For the implementation, we decided to adopt the same problem to see the difference between languages clearly. This also gives some hints about the decisions we made while creating 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 traces between the elements of source and target languages. This pattern was originally proposed in [6] and later refined in [21].

– **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 src.r1(src.e, src.f), trgt.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, src.r1, t1, t2, ~trgt.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:**
  - **Henshin:** We show the implementation of ER mapping in Henshin in Fig. 2.

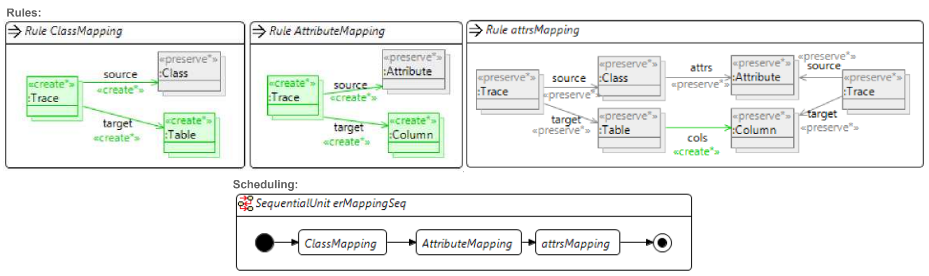


Fig. 2. Rules of ER Mapping in Henshin

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: the classMapping for mapping classes to tables and the attributeMapping for mapping attributes to columns. There is also one rule for the relationMapping which is the attrsRelationMapping rule maps the attribute relation of a class to the column relation of a table to related the table with its corresponding column. In Henshin, trace is a separate class with source and target links to the elements it will connect. 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.

- **MoTif:** Fig. 3 depicts the implementation of ER Mapping in MoTif.

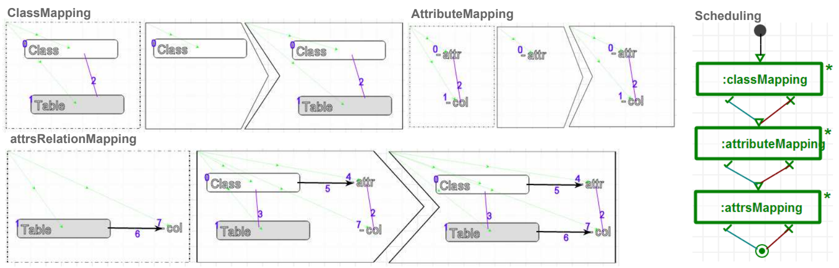


Fig. 3. Scheduling and Rules of ER Mapping in MoTif

The structure of MoTif rules are similar to the one in the design pattern. Also MoTif provides a graphical syntax with seperate parts for LHS and RHS and NACs unlike Henshin. One can see the NAC elements in a dashed rectangle, LHS elements at the right of the arrow structure and RHS at the left of the arrow structure. The trace links in MoTif are designed as a separate formalism and imported whenever we need. The scheduling, shown on the right of Fig. 3, is a dedicated control flow structure in MoTif with a start and end node. For this design pattern, the three rules are applied in sequence by connecting the success and fail ports to the input port of the next rule. Also, these rules are the SRule structures in MoTif which represents the recursive application and has a $*$ on top right.

- **AGG:** AGG rules are depicted in Fig. 4. The syntax of AGG rules is very similar to the syntax of MoTif 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. For the scheduling of the AGG rules, to give the sequence meaning, the layers are set for each rule. The classMapping, the attributeMapping, and the attrsRelationMapping rules have layers $0, 1, 2$ respectively, which encodes the desired scheduling.
- **Viatra2:**

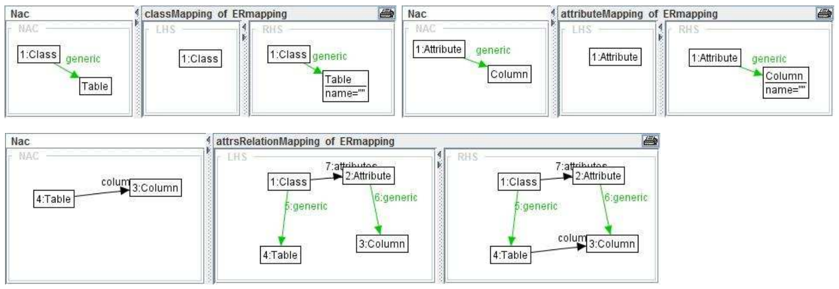
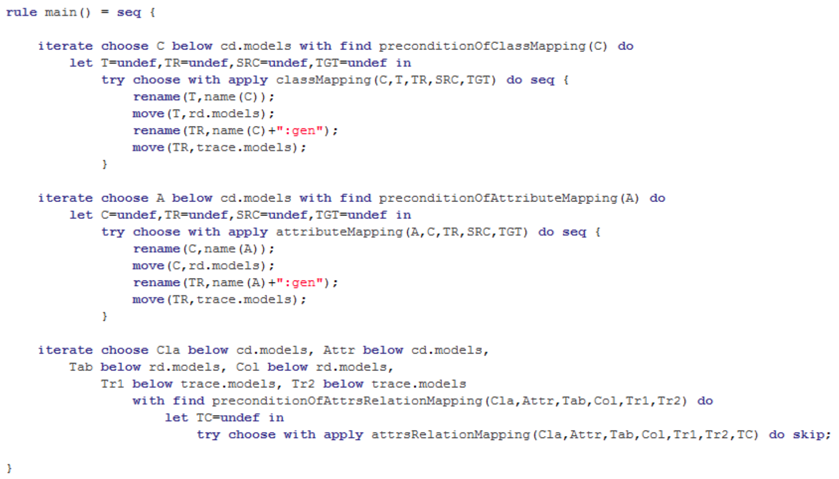


Fig. 4. Rules of ER Mapping in AGG

Viatra2 provides a textual environment to represent the rules and the scheduling. Fig. 5 depicts the rules and the necessary patterns for the rules. In Viatra2, one can adopt pattern structure for reusability purposes. Each rule is given with a gtrule keyword and can consist of precondition pattern, postcondition pattern and action. Other patterns can be called inside pre and post condition patterns and new elements can be created in action. Each rule is calling the patterns provided above of the rules. Traces are explicitly created in Viatra2. Therefore each element we want to connect with a trace must have a corresponding trace element. For example class2table element is a trace to connect class with a table. Negative patterns can be called with neg keyword.



```
rule main() = seq {

    iterate choose C below cd.models with find preconditionOfClassMapping(C) do
        let T=undef,TR=undef,SRC=undef,TGT=undef in
            try choose with apply classMapping(C,T,TR,SRC,TGT) do seq {
                rename(T,name(C));
                move(T,rd.models);
                rename(TR,name(C)+":gen");
                move(TR,trace.models);
            }

    iterate choose A below cd.models with find preconditionOfAttributeMapping(A) do
        let C=undef,TR=undef,SRC=undef,TGT=undef in
            try choose with apply attributeMapping(A,C,TR,SRC,TGT) do seq {
                rename(C,name(A));
                move(C,rd.models);
                rename(TR,name(A)+":gen");
                move(TR,trace.models);
            }

    iterate choose Cla below cd.models, Attr below cd.models,
        Tab below rd.models, Col below rd.models,
            Tr1 below trace.models, Tr2 below trace.models
                with find preconditionOfAttrsRelationMapping(Cla,Attr,Tab,Col,Tr1,Tr2) do
                    let TC=undef in
                        try choose with apply attrsRelationMapping(Cla,Attr,Tab,Col,Tr1,Tr2,TC) do skip;

}
```

Fig. 6. Scheduling of ER Mapping in Viatra2

The scheduling of the Viatra2 rules is depicted in Fig. 6. Viatra2 starts the rule execution with a main block. Inside the block, each rule may be applied

```
pattern isClassGenericConnected(C) =
{
    class(C);
    table(T);
    class2table(TR);
    class2table.src(SRC,TR,C);
    class2table.trgt(TGT,TR,T);
}

pattern preconditionOfClassMapping(C) =
{
    class(C);
    neg find isClassGenericConnected(C);
}

gtrule classMapping(in C,out T,out TR,out SRC,out TGT) =
{
    precondition pattern lhs(C) =
    {
        find preconditionOfClassMapping(C);
    }
    action
    {
        new(table(T));
        new(class2table(TR));
        new(class2table.src(SRC,TR,C));
        new(class2table.trgt(TGT,TR,T));
    }
}
```

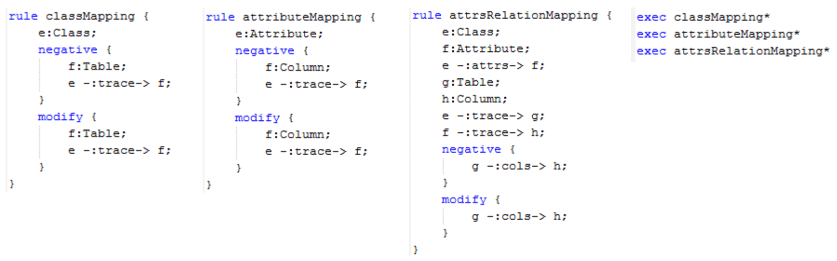
```
pattern isAttributeGenericConnected(A) =
{
    attribute(A);
    column(C);
    attr2col(TR);
    attr2col.src(SRC,TR,A);
    attr2col.trgt(TGT,TR,C);
}

pattern preconditionOfAttributeMapping(A) =
{
    attribute(A);
    neg find isAttributeGenericConnected(A);
}

gtrule attributeMapping(in A,out C,out TR,out SRC,out TGT) =
{
    precondition pattern lhs(A) =
    {
        find preconditionOfAttributeMapping(A);
    }
    action
    {
        new(column(C));
        new(attr2col(TR));
        new(attr2col.src(SRC,TR,A));
        new(attr2col.trgt(TGT,TR,C));
    }
}
```

```
pattern isTableAndColumnConnected(T,C) =
{
    table(T);
    column(C);
    table.cols(TC,T,C);
}

pattern preconditionOfAttrsRelationMapping(Cla,Attr,Tab,Col,Tr1,Tr2) =
{
    class(Cla);
    attribute(Attr);
    table(Tab);
    column(Col);
    class2table(Tr1);
    class2table.src(Src1,Tr1,Cla);
    class2table.trgt(Trgt1,Tr1,Tab);
    attr2col(Tr2);
    attr2col.src(Src2,Tr2,Attr);
    attr2col.trgt(Trgt2,Tr2,Col);
    neg find isTableAndColumnConnected(Tab,Col);
}

gtrule attrsRelationMapping(in Cla, in Attr, in Tab, in Col, in Tr1, in Tr2,out TC) =
{
    precondition pattern lhs(Cla,Attr,Tab,Col,Tr1,Tr2) =
    {
        find preconditionOfAttrsRelationMapping(Cla,Attr,Tab,Col,Tr1,Tr2);
    }
    action
    {
        new(table.cols(TC,Tab,Col));
    }
}
```

Fig. 5. Rules of ER Mapping in Viatra2

one by one to represent the sequence. Each variable has to be defined at the beginning of a rule call block. The three rules are called sequentially in the figure. After each rule is applied successfully, then one can modify the newly created elements within a do block. In the figure, we are renaming the newly created elements and moving them to the correct location in the hierarchy. In the application of attrsRelationMapping rule, we are just modifying an attribute in tables (cols attribute), so we do not need to do anything in the do block.

- **GrGen.NET:**
  GrGen.NET rules are similar to the design pattern rules with only the necessary elements within. The rules are depicted in Fig. 7.



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

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

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

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

Fig. 7. Rules of ER Mapping in GrGen.NET

The constraint elements are defined in the entry of the rule. NACs are supported with negative block in GrGen.NET. The scheduling is also depicted at the right of the Fig. 7, which is just a sequential and recursive application of the rules. We use the same notation for the recursive rule application in DelTa with GrGen.NET.

– **Variations:** Sometimes the entities in specific metamodels can not 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 as depicted in Listing 1.3.

Listing 1.3. One-to-Many Entity Relationship Mapping MTDP

```
mtdp OneToOneERMapping
    metamodels: src, trgt
    rule entityMapping*
        Entity src.e
        ElementGroup trgt.eg
        Trace t1(src.e, trgt.eg)
        constraint: src.e, ~trgt.eg, ~t1
        action: trgt.eg, t1
    rule relationMapping*
        Entity src.e, src.f
        ElementGroup trgt.eg1, trgt.eg2
        Relation src.r1(src.e, src.f), trgt.r2(trgt.eg1, trgt.eg2)
        Trace t1(src.e, trgt.eg1), t2(src.f, trgt.eg2)
        constraint: src.e, src.f, trgt.eg1, trgt.eg2, src.r1, t1, t2, ~trgt.r2
        action: r2
    Sequence: START, entityMapping, relationMapping, END
```

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]. It generates the intermediate paths between nodes that are not necessarily directly connected via trace 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 1.4. 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 1.4. 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:**
  • **AGG:** We have implemented the transitive closure in AGG. Fig. 8 depicts the

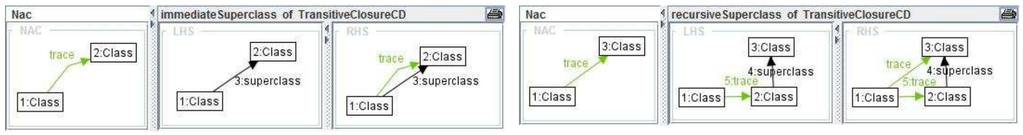


Fig. 8. Transitive Closure rules in AGG

corresponding rules. The immediateSuperclass rule creates a trace between a class and its superclass. The NAC prevents this trace from being created again. The recursiveSuperclass rule creates the remaining trace 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.

- **MoTif:** The rules and the scheduling for MoTif are depicted in Fig. 9. Each

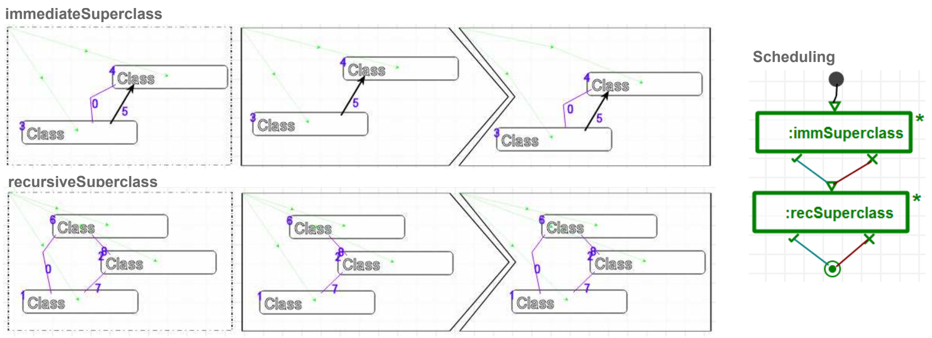


Fig. 9. Transitive Closure rules and scheduling in MoTif

rule is recursive rule since we want to apply to all matches. The purple links in the rules are trace links. In the scheduling part of the figure, one can see the sequence structure of the two rules.

- **Henshin:** The rules and scheduling structures for Henshin are depicted in Fig. 10. Henshin provides some predefined units for scheduling. Also what Henshin can not do with rules are setting them as recursive. For that reason, we have used one of predefined structures of Henshin, which is LoopUnit, to provide the recursive meaning for the recursiveSuperclass rule. Then, the immediateSuperclass and the recursiveSuperclass rules are put inside a SequentialUnit as stated in the design pattern.

- **Viatra2:**

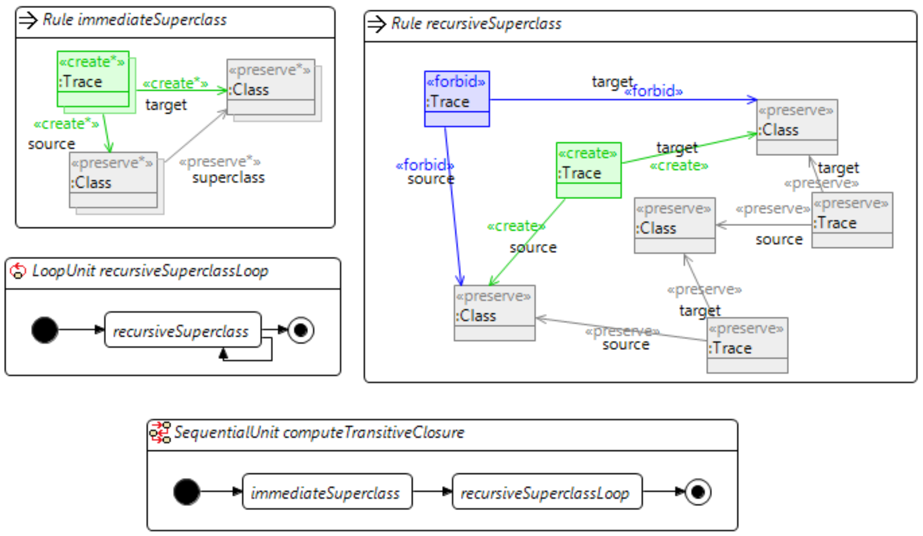


Fig. 10. Transitive Closure rules and scheduling in Henshin

```
pattern preconditionOfImmediateSuperclass(N1, N2) =
{
    class(N1);
    class(N2);
    class.superclass(NXT,N1,N2);
    neg find isGenericConnected(N1,N2);
}

pattern isGenericConnected(N1, N2) =
{
    class(N1);
    class(N2);
    class2class(T);
    class2class.src(SRC,T,N1);
    class2class.target(TGT,T,N2);
}

gtrule immediateSuperclass(in N1, in N2, out T, out SRC, out TGT) =
{
    precondition pattern lhs(N1, N2) =
    {
        find preconditionOfImmediateSuperclass(N1,N2);
    }
    action
    {
        new(class2class(T));
        new(class2class.src(SRC,T,N1));
        new(class2class.target(TGT,T,N2));
    }
}
```

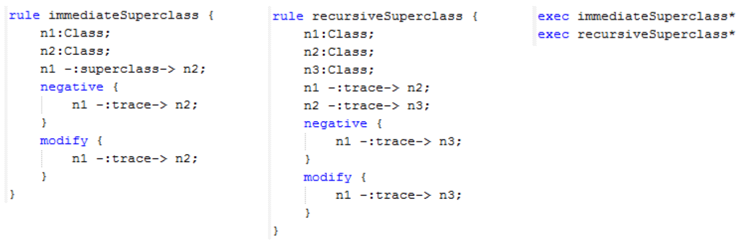
```
pattern preconditionOfRecursiveSuperclass(N1,N2,N3,T1,T2) =
{
    class(N1);
    class(N2);
    class(N3);
    class2class(T1);
    class2class(T2);
    class2class.src(SRC1,T1,N1);
    class2class.target(TGT1,T1,N2);
    class2class.src(SRC2,T2,N2);
    class2class.target(TGT2,T2,N3);
    neg find isGenericConnected(N1,N3);
}

gtrule recursiveSuperclass(in N1, in N2, in N3, in T1, in T2,
    out T3, out SRC3, out TGT3) =
{
    precondition pattern lhs(N1,N2,N3,T1,T2) =
    {
        find preconditionOfRecursiveSuperclass(N1,N2,N3,T1,T2);
    }
    action
    {
        new(class2class(T3));
        new(class2class.src(SRC3,T3,N1));
        new(class2class.target(TGT3,T3,N3));
    }
}
```

```
    rule main() = seq {

        iterate choose N1 below tree.models, N2 below tree.models with find preconditionOfImmediateSuperclass(N1, N2) do
            let T=undef,SRC=undef,TGT=undef in
                try choose with apply immediateSuperclass(N1,N2,T,SRC,TGT) do seq {
                    rename(T,"trace("+name(N1)+":"+name(N2)+")");
                    move(T,treeTrace.models);
                }

        iterate choose N1 below tree.models, N2 below tree.models, N3 below tree.models,
            T1 below treeTrace.models, T2 below treeTrace.models
                with find preconditionOfRecursiveSuperclass(N1,N2,N3,T1,T2) do
                    let T3=undef,SRC3=undef,TGT3=undef in
                        try choose with apply recursiveSuperclass(N1,N2,N3,T1,T2,T3,SRC3,TGT3) do seq {
                            rename(T3,"trace("+name(N1)+":"+name(N3)+")");
                            move(T3,treeTrace.models);
                        }
    }
```

Fig. 11. Transitive Closure rules and scheduling in Viatra2

The rules and scheduling for Viatra2 are depicted in Fig. 11. The two rules are using some patterns for helpers. The isGenericConnected pattern checks if two nodes are connected with a trace class, which is called class2class and has source and target links. This pattern is used a NAC in the precondition of the rules.

- **GrGen.NET:**



```
rule immediateSuperclass {
    n1:Class;
    n2:Class;
    n1 -:superclass-> n2;
    negative {
        n1 -:trace-> n2;
    }
    modify {
        n1 -:trace-> n2;
    }
}
```

```
rule recursiveSuperclass {
    n1:Class;
    n2:Class;
    n3:Class;
    n1 -:trace-> n2;
    n2 -:trace-> n3;
    negative {
        n1 -:trace-> n3;
    }
    modify {
        n1 -:trace-> n3;
    }
}
```

```
exec immediateSuperclass*
exec recursiveSuperclass*
```

Fig. 12. Transitive Closure rules and scheduling in GrGen.NET

The rules for GrGen.NET and execution of the rules are depicted in Fig. 12. The modify block is the action of the GrGen.NET rules. In the execution, we also put the * character to make them recursive as stated in the design pattern.

– **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.

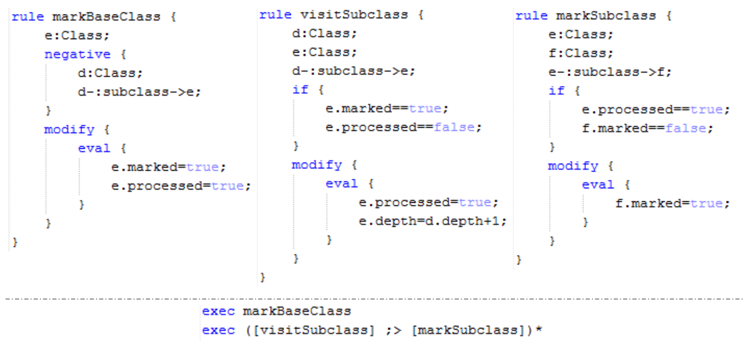
– **Structure:**

Listing 1.5. 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
    Success: markNextEntity, visitEntity
    Fail: markNextEntity, END
```

As depicted in Listing 1.5, 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.
– **Implementation:**
  • **GrGen.NET:** We have implemented visitor in GrGen.NET as depicted in Fig. 13.



```
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])*
```

Fig. 13. Visitor rules and scheduling in GrGen.NET

This MTL provides a textual syntax for both rules and scheduling mechanisms. In a rule, the constraint is defined by declaring the elements of the pattern and conditions on attributes are checked with an if statement. Actions are written in a modify or replace statement for new node creation and eval statements are used for attribute manipulation. The markBaseClass rule selects a class with no superclass as the initial element to visit. 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. 13. 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.

- **MoTif:** The rules in MoTif and the scheduling of these rules are depicted in

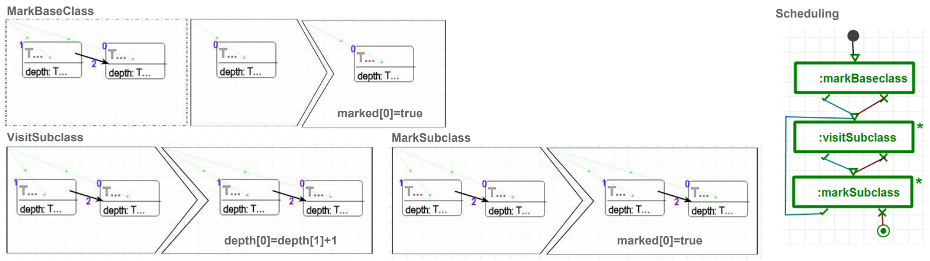


Fig. 14. Visitor rules and scheduling in MoTif

Fig. 14. MoTif lets writing actions (which are similar to the action part in DelTa but a more pragmatic version) to be executed after RHS is applied. The texts at the bottom of the RHS in the rules are these actions. For example, in the MarkBaseClass rule, the marked attribute of the element 0 is set to true. In the VisitSubclass rule, the action is setting the depth level of the element 0 one more than the element 1. As one can see, in MoTif, we can easily create loop structures by just connecting one of the output ports (success or fail) of a rule to the input of another rule, *e.g.*, we connected the success port of the markSubclass rule to the input port of the visitSubclass rule. In addition to the SRule, the markBaseclass rule is an ARule in terms of MoTif structures, which represents a rule with application count 1.

- **AGG:** The rules for AGG are depicted in Fig. 15. The structure of the rules to-

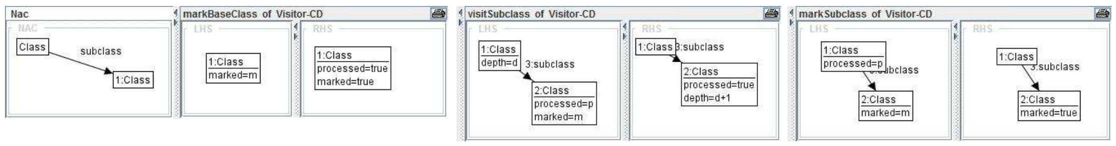


Fig. 15. Visitor rules in AGG

tally fits with the other languages. Visitor requires a more sophisticated scheduling structure (*i.e.*, looping), which AGG does not have. For that reason, we put the markBaseClass rule in layer 0 and the other two rules in layer 1. Since these are in the same layer, each rule may apply nondeterministically whenever applicable, which does not violate the main idea of the design pattern.

- **Henshin:** The rules for Henshin are depicted in Fig. 16. In Henshin, the at-

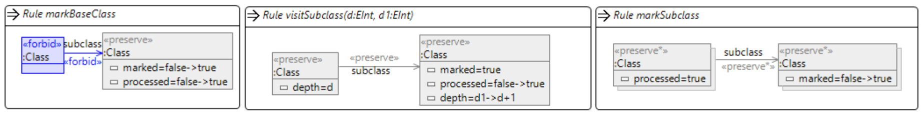


Fig. 16. Visitor rules in Henshin

tribute values of a class are passed along by putting them as parameters to the rule. In the visitSubclass rule, the d parameter is to hold the depth of the first class and the d1 parameter is to hold the depth of the second class. Then the new depth value of the second class is calculated by increasing d by 1. The schedul-

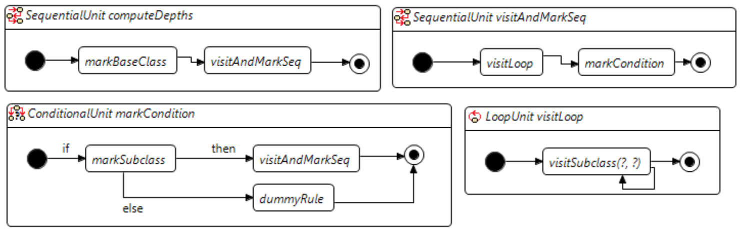


Fig. 17. Scheduling of visitor rules in Henshin

ing of these rules are depicted in Fig. 17. The main scheduling starts with the computeDepths sequential unit, by executing the markBaseClass rule and calling the visitAndMarkSeq sequential unit. The visitAndMarkSeq unit first calls the visitLoop, which executes the visitSubclass rule in a loop, and then calls the markCondition conditional unit. The markCondition unit tries to execute the markSubclass rule and in case of a success, it calls the visitAndMarkSeq again to go for processing another set of marked classes. If it fails, then the transformation finish with a dummyRule. Henshin does not accept empty blocks for if-then-else structure, so we needed that empty dummy rule.

– **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].
– **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 1.6. 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
    Success: checkFixedPoint, END[result=true]
    Fail: checkFixedPoint, iterate
    Success: iterate, checkFixedPoint
```

The structure is depicted in Listing 1.6. 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 checkFxedPoint 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:**
  - **MoTif:** Fig. 18 shows the implementation of the LCA from [5] in MoTif using the fixed point iteration pattern. The initiate rule is extended to create traces links on the input nodes themselves with the LinkToSelf rules and with their parents with the LinkToParent rules. In MoTif, these two rules are put inside a container called CRule, which is represented with double lined rectangle. The GetLCA rule implements the checkFixedPoint rule and tries to find the LCA of the two nodes in the resulting model following trace links. This rule is a

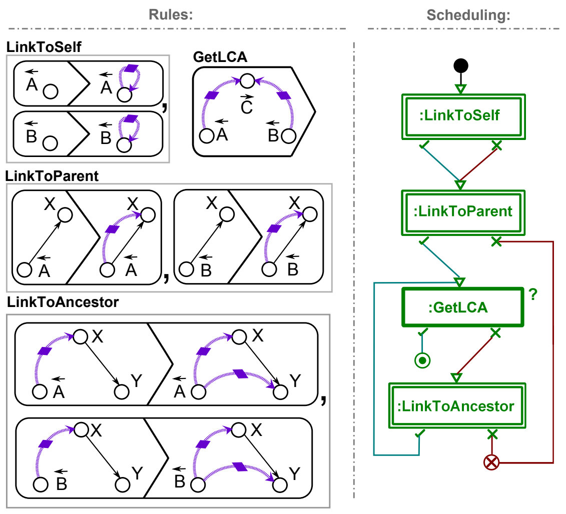


Fig. 18. Rules and Scheduling in MoTif

QRule, represented with a question mark on top right and 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 1.6.

- **AGG:** The rules for computing LCA in AGG are depicted in Fig. 19.

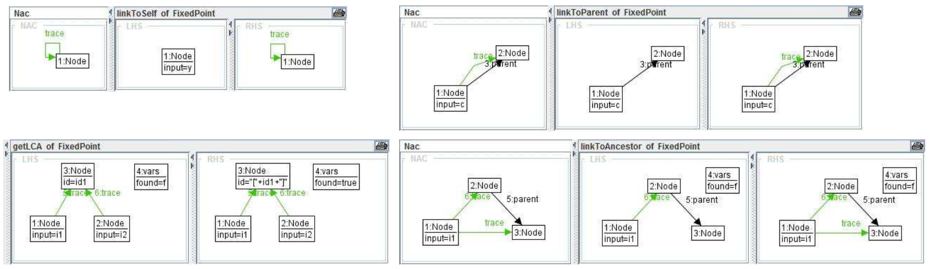


Fig. 19. Rules of Fixed Point Iteration in AGG

The rules are almost same as MoTif rules. The lack of sophisticated scheduling constructs also lead us to do tricks to realize LCA. For that reason, the initiate rules' layer are set to 0. The getLCA and the linkToAncestor rules' layer are set to 1, so that check step can be done and whenever it fails, the execution can go with the linkToAncestor rule. If both fails, then the execution finishes.

- **Henshin:** The rules to compute LCA in Henshin are depicted in Fig. 20. The scheduling of these rules are depicted in Fig. 21. Since Henshin does not pro-

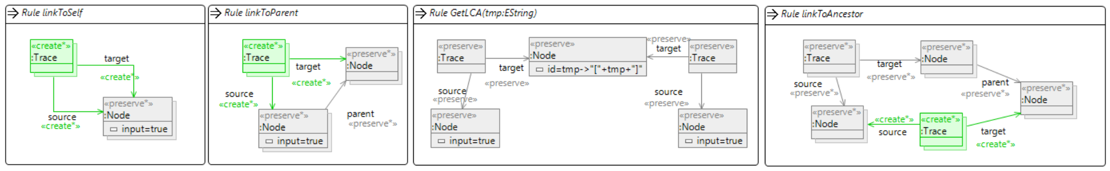


Fig. 20. Rules of Fixed Point Iteration in Henshin

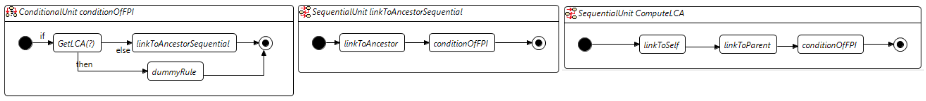
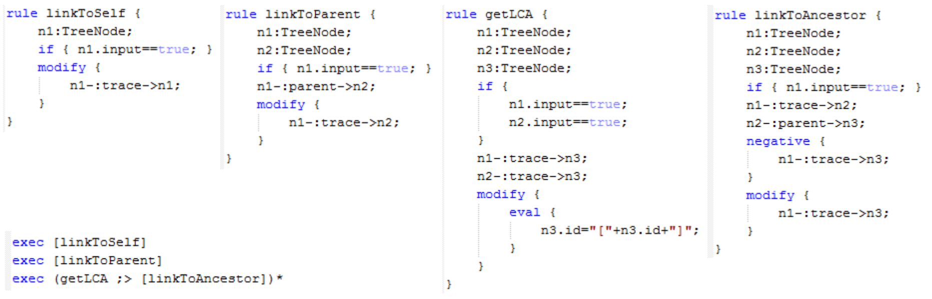


Fig. 21. Scheduling of the rules of Fixed Point Iteration in Henshin

vide an exact activity diagram like structure for scheduling, we need to use some additional structures to get the exact scheduling stated in the design pattern. The ComputeLCA sequential unit is the main scheduling of the transformation. As can be seen in figure, the rules of initiate, which are the linkToSelf and the linkToParent, are sequentially applied and then we go to the conditionOfFPI conditional unit. The conditionOfFPI unit is a simple if-then-else structure and consist of only one rule or only one unit in each block. The condition is the GetLCA rule. Success of the condition means the transformation can end. As explained in the Henshin implementation of Visitor pattern, we called an empty dummyRule, since Henshin does not accept empty blocks. Failure of the condition means another execution of the iterate rule in the design pattern, which we call the linkToAncestorSequential sequential unit. This unit executes the linkToAncestor rule and calls the conditionOfFPI again for another check of the fixed point.

- **GrGen.NET:** The rules and the scheduling of computing LCA in GrGen.NET are depicted in Fig. 22.



```
rule linkToSelf {
    n1:TreeNode;
    if { n1.input==true; }
    modify {
        n1-:trace->n1;
    }
}
```

```
rule linkToParent {
    n1:TreeNode;
    n2:TreeNode;
    if { n1.input==true; }
    n1-:parent->n2;
    modify {
        n1-:trace->n2;
    }
}
```

```
rule getLCA {
    n1:TreeNode;
    n2:TreeNode;
    n3:TreeNode;
    if {
        n1.input==true;
        n2.input==true;
    }
    n1-:trace->n3;
    n2-:trace->n3;
    modify {
        eval {
            n3.id="["+n3.id+"]";
        }
    }
}
```

```
rule linkToAncestor {
    n1:TreeNode;
    n2:TreeNode;
    n3:TreeNode;
    if { n1.input==true; }
    n1-:trace->n2;
    n2-:parent->n3;
    negative {
        n1-:trace->n3;
    }
    modify {
        n1-:trace->n3;
    }
}
```

```
exec [linkToSelf]
exec [linkToParent]
exec (getLCA ;> [linkToAncestor])*
```

Fig. 22. Rules and Scheduling of Fixed Point Iteration in GrGen.NET

The attributes can be modified in GrGen.NET rules with eval block inside the modify block, *e.g.,* id attribute in n3 is modified in the getLCA rule. The scheduling of the rules is depicted at the bottom left of the figure. The linkToSelf and the linkToParent rules are executed once and for all matches. Then the GetLCA and the linkToAncestor are connected sequentially with `;>` sign, and made recursive together with the `*` character, which means the recursive part will succeed when the linkToAncestor rule succeeds.

– **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].

4 Conclusion

In this technical report, we revisited the syntax and semantics of DelTa, described four known design patterns for model transformation in DelTa and implemented them in five different languages.

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.

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., Vangheluwe, H.: Modeling a Model Transformation Language. In: Domain Engineering: Product Lines, Conceptual Models, and Languages. Springer (2012)
9. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS. Springer-Verlag (2006)

10. 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
11. Syriani, E., Vangheluwe, H.: A Modular Timed Model Transformation Language. Journal on Software and Systems Modeling **12**(2) (jun 2011) 387–414
12. Agrawal, A., Karsai, G., Shi, F.: Graph transformations on domain-specific models. Journal on Software and Systems Modeling (2003)
13. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Science of Computer Programming **68**(3) (2007) 214–234
14. Syriani, E., Vangheluwe, H., LaShomb, B.: T-Core: a framework for custom-built model transformation engines. Software & Systems Modeling (2013) 1–29
15. 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
16. Czarnecki, K., Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal **45**(3) (jul 2006) 621–645
17. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: AGTIVE. Springer (2004) 446–453
18. 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
19. Syriani, E., Vangheluwe, H.: De-/Re-constructing Model Transformation Languages. EASST **29** (mar 2010)
20. 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
21. Kevin Lano, Shekoufeh Kolahdouz Rahimi: Constraint-based specification of model transformations. Journal of Systems and Software **86**(2) (2013) 412–436