Model-transformation Design Patterns

K. Lano, S. Kolahdouz-Rahimi
Dept. of Informatics, King's College London

Abstract—This paper defines a catalogue of patterns for the specification and design of model transformations, and provides a systematic scheme and classification of these patterns, together with pattern application examples in leading model transformation languages such as ATL, QVT, GrGen.NET, and others. We consider patterns for improving transformation modularization and efficiency and for reducing data storage requirements.

We define a metamodel-based formalization of model transformation design patterns, and measurement-based techniques to guide the selection of patterns. We also provide an evaluation of the effectiveness of transformation patterns on a range of different case studies.

Index Terms—Model transformations; design patterns; model-driven development.

1 Introduction

Model-Driven Development (MDD) is the development of software systems by means of the construction and transformation of models [37]. MDD can be used to rapidly develop software from high-level specifications, by means of the transformation of abstract models into more refined models, and by the automated generation of executable code from models. MDD was devised to make the production of software systems more reliable and efficient, and to retain the core functionality of a system despite changes in its technology.

Transformations are an essential part of model-driven development, enabling system descriptions at one level of abstraction to be mapped into descriptions at a lower or higher level (refinement and abstraction transformations), supporting the migration of models from one modeling language or notation to another, and supporting the refactoring and analysis of models. With the increasing scale and complexity of models utilized within software development, there has been a consequent increase in the scale and complexity of model transformations. Systematic and rigorous approaches for model transformation development are therefore required [23].

Design patterns for model transformations have been introduced [1], [8], [11], [14], [26], [29] to provide solutions for a number of model transformation specification and design problems, and to improve the quality of model transformation specifications and designs, in particular, (i) to enable the factoring of complex transformations into modular sub-transformations; (ii) to simplify individual mapping rules of a transformation; (iii) to improve the efficiency of a transformation by removing redundant and duplicated evaluations, optimizing execution strategies and simplifying complex model navigations.

In [43], [46] and [48] we briefly described several transformation patterns for the UML-RSDS language, and introduced techniques for their application and selection using metrics. In this paper we give a complete and comprehensive pattern catalog, for model transformation languages in general, with detailed descriptions of patterns, their inter-relationships and variations. We provide a detailed evaluation of the benefits of transformation patterns, and a formal framework for their semantics.

In Sections 2 and 3 we summarize the concepts of model transformations and design patterns. Section 4 gives an overview of model transformation design patterns. A detailed catalogue of model transformation design patterns is given in Sections 5, 6, 7, 8 and 9. In Section 10 we identify what combinations of patterns can be used together. Section 11 gives metrics and heuristics to select patterns. Sections 12 and 13 give extended examples of applying transformation patterns, Section 14 evaluates the benefits of applying transformation patterns, and Section 15 describes related work. In the appendix we give a metamodelling semantics for model transformations and transformation design patterns.

Our particular contributions are to give a systematic presentation of a wide range of patterns, to define metrics to guide their selection, and to define a formal framework for transformation patterns.

2 Model Transformations

Model transformations operate to produce one or more new models from one or more existing models, or to update models in-place, for a range of purposes within MDD, for example to improve the quality of a model or to refine a model towards an executable system, to migrate a model from one language to another language, or to compare two models.

Figure 1 shows the typical context of a model transformation $\tau : S \to T$ from a source language S to a target language T. τ is typically specified at the language level in terms of the entity types and features of S and T.

Examples of transformation specification notations are Triple Graph Grammars [65] and UML-RSDS [47]. Transformations also have implementations, derived from the specifications, and these implementations operate at the model level upon instances (models) m of S as inputs and produce instances n of T as outputs. S and T may be modeling languages such as UML, programming languages such as Java, or domain-specific languages, e.g., describing electronic health records or financial transactions. It is usual to define S and T by metamodels, in this paper we will use a subset $\mathcal{LMM}$ of the UML class diagram metamodel to define languages (Figure 1 in the appendix). Transformations may in general have multiple input and output models and languages, so that the general form of a transformation is

$$\tau : SL_1 \times ... \times SL_n \to TL_1 \times ... \times TL_m$$

In this paper we will focus on the case $n = 1$ and $m = 1$.

Transformations themselves can be defined by a metamodel $\mathcal{TSPEC}$ (Figure 3 in the appendix), which is based upon $\mathcal{LMM}$. Transformation specifications usually consist of a set or sequence of *transformation rules* operating on one or more source or target language entity types.

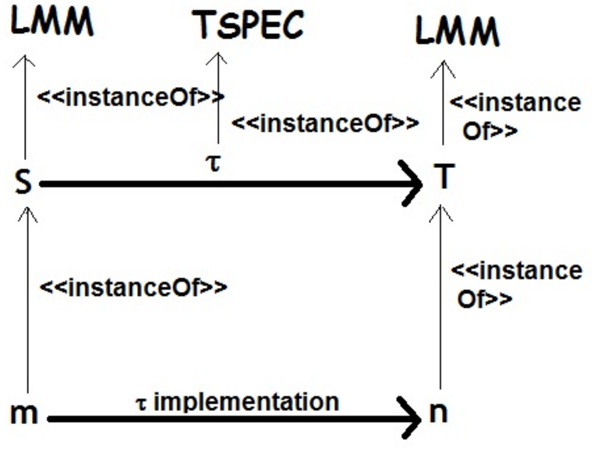

Fig. 1. Model transformation architecture

Throughout the paper we will use UML class diagram notation to visually represent languages, and UML object diagram notation to express models. Activity diagrams will be used to express compositions of transformations within a system of transformations. Transformations will be visually defined using constraint annotations and dependency relationships on class and object diagrams, and textually defined using particular transformation languages, such as QVT-R [53], GrGen [27] or UML-RSDS [47]. Courier font text will be used for code in each particular transformation language. We will also use a generalized transformation specification notation using a textual representation, based on $\mathcal{TSPEC}$, in order to express patterns in a language-independent manner. Mathematical font will be used for such specifications. Rules in this representation have the forms (1):

for each $s1 : S1;\ ...;\ sn : Sn$ **satisfying** *SCond*
create $t1 : T1;\ ...;\ tm : Tm$ **satisfying** *Post*

which create target objects $t1$ to tm for each tuple of source objects $s1$ to sn satisfying *SCond*, and establish the predicate *Post* between the si and tj, or (2):

for each $s1 : S1;\ ...;\ sn : Sn$ **satisfying** *SCond*
do *Post*

which performs updates to the source or target model to establish *Post* for the si. Form (1) is equivalent to

for each $s1 : S1;\ ...;\ sn : Sn$ **satisfying** *SCond*
do $T1 \rightarrow exists(t1 \mid ...\ Tm \rightarrow exists(tm \mid\ Post)\ ...\ )$

in the second form[1]. The **create unique** keyword indicates that no new objects $t1, ..., tm$ should be created satisfying *Post* if such objects already exist, in form (1). A **create unique** clause corresponds to $\rightarrow exists1$ quantification.

A form of rule which simply deletes elements that satisfy a condition is (3):

for each $s1 : S1$ **satisfying** *SCond*
delete $s1$

This is equivalent to

for each $s1 : S1$ **satisfying** *SCond*
do $s1 \rightarrow isDeleted()$

Model transformations are often categorized as *endogenous* (where the source and target languages are the same) or *exogenous* (for different source and target languages) [13]. Transformations can be *update-in-place*, where the source model is modified by the transformation, or can work with separate source and target models. *Input-preserving* transformations do not modify their source model, and usually populate an initially empty target model. Exogenous transformations include *migrations* (mapping models of one version of a language to models of another version), *refinements* (mapping a model at a higher level of abstraction to one at a lower level), and *abstractions* (the inverse of refinements). Such transformations are also termed *model translations* [64]. Update-in-place transformations include *restructuring*, *refactoring* and *quality-improvement* transformations, which modify a model to improve some quality measure or to impose some property on the model. These are also termed *model rephrasings* [64]. Transformations are *unidirectional* if they can only operate in a single direction (from source to target) and *bidirectional* if they can operate also in the reverse direction.

Transformations can be specified or implemented using a conventional programming language, such as Java,

1. $E \rightarrow exists(t \mid P)$ abbreviates $E.allInstances() \rightarrow exists(t \mid P)$ for entity types E. $E \rightarrow forAll(t \mid P)$ abbreviates $E.allInstances() \rightarrow forAll(t \mid P)$.

or by using a special-purpose transformation language, such as Kermeta [32], ATL [30], QVT [53] or ETL [34]. Such languages provide a specialized syntax and language constructs to define transformations as collections of transformation rules. Transformation languages can be characterized as being imperative (Kermeta, QVT-O), declarative relational (QVT-R), hybrid relational (ATL, ETL, UML-RSDS [41]) or declarative/hybrid graph-transformation (TGG, GrGen.NET [27], GReAT [6], Viatra [56]).

Various issues need to be considered for the development of model transformations. The area of transformation specification, design and implementation lacks well-established techniques for ensuring quality in transformations. Transformation design patterns can potentially assist in the systematic construction of model transformations. For transformations (such as refinements and migrations) without any update-in-place behavior, modularization patterns (Section 5) can help to organize the transformation specification in a systematic, well-structured and clear manner. For update-in-place transformations, the specification task is often more challenging, and patterns such as Parallel Composition (Section 5.5) can be used to minimize the parts of the transformation which require implementation techniques such as fixed-point iteration, for example. Optimization patterns from Section 6 can be used to improve the efficiency of either input-preserving or update-in-place transformations.

A technical property of transformations which is useful to support the application of modularization patterns is the following. In cases where transformation rules are applied in defined orders, the transformation should satisfy the condition that if rule $R1$ refers to instances of an entity type $T2$, then any other rule $R2$ which creates $T2$ instances should precede $R1$. More precisely, for each rule R define the *write frame* $wr(R)$ of R, and the *read frame* $rd(R)$. These are the sets of entity types and features which R may update or access, respectively.

A dependency ordering $Rn < Rm$ is defined between distinct rules by

$$wr(Rn) \cap rd(Rm) \neq \{\}$$

"Rm depends on Rn".

Then a transformation with rules ordered as $R_1, \ldots, R_n$ should satisfy the *syntactic non-interference* conditions:

1) If $R_i < R_j$ and $i \neq j$, then $i < j$.
2) If $i \neq j$ then $wr(R_i) \cap wr(R_j) = \{\}$.

Together, these conditions ensure that subsequent rules R_j cannot invalidate earlier rules R_i, for $i < j$.

If a rule r has $rd(r)$ disjoint from $wr(r)$ then it can usually be implemented by a bounded iteration (such as a *for*-loop over a fixed set of elements). Otherwise a fixed-point iteration may be required, in which the rule is applied until no more input elements exist that match its application conditions.

3 Design Patterns

The field of software design patterns has grown extensively since the first work on software patterns in the 1990s [18]. Design patterns have proved useful as encodings of good design practice and expert knowledge in a wide variety of domains, such as enterprise information systems [37], service-oriented architectures and software security [24].

A design pattern expresses a characteristic means of solving a common design problem: the pattern describes the software structures and elements, such as classes, objects and methods, that constitute the solution idea. It is also important to include a description of the problem which motivated the pattern, and how such problem situations can be detected. It is important also not only to consider the benefits of a particular pattern, but any negative consequences and reasons against introducing the pattern.

Patterns can in some cases be considered as forms of transformations: the application of a pattern rewrites a software model or system with a problematic structure into a model or system with an improved structure. For example, Template Method replaces multiple subclass methods which have duplicated code by a new superclass method which factors out the duplicated code, and subclass hook methods which contain the variation points [18]. However, the automatic selection and application of patterns is still impractical in general due to the variability of pattern instantiations and the need for human expertise in selecting appropriate patterns and applying them.

Patterns can be distinguished from *idioms*: small-scale repetitive structures of code, such as the standard for-loop header in Java. Patterns are also distinguished from *generic modules*, which define a template for a family of components obtainable by instantiating the parameters of the generic module. The concept of program and design *refactoring* is closely related to design patterns [16]: refactorings can be considered to be incremental changes to the structure of a system (such as promoting an attribute from subclasses up to a superclass) which improve some quality measure of the structure. In contrast, design patterns often involve more radical changes in structure, also with the intention of improving some quality measure, but operating at a higher semantic level, which is less easily automated. For example, the Auxiliary Metamodel pattern (Section 5.7) defines a general strategy for reducing the complexity of a transformation specification, but there is potentially great variability in the choice of how this strategy is carried out. Refactorings can be used as steps towards a pattern introduction [31].

Various proposals have been made for formalizing and verifying design patterns, e.g., [7], [36]. In the appendix we express transformation patterns as (meta) transformations operating upon the transformation specification metamodel $\mathcal{TSPEC}$. The patterns we describe here will usually preserve the original semantics of the transformation, considered as a mapping from the original source language to the original target language. They will also usually improve one or more quality measures, e.g., to reduce the average rule syntactic complexity in the case of rule modularization patterns, or the worst case computational complexity of transformation implementations in the case of optimization patterns.

4 Design Patterns for Model Transformations

We define a model transformation design pattern as, "A general repeatable solution to a commonly-occurring model transformation design problem" [26], and likewise for specification patterns. Patterns for model transformations have been proposed by several researchers [1], [8], [11], [14], [26], [29], [43]. These apply the general concept of software design pattern to the model transformation domain. Since design patterns can improve the flexibility, reusability and comprehensibility of software systems, the same benefits should hold for model transformation patterns. Specialized issues for model transformations are: (i) the need to inspect and modify highly complex data (models), and to partition large models for separate processing [66]; (ii) the need to control and schedule transformation steps, and (iii) to modularize transformations to optimize flexibility and reuse, taking into account that evolution of metamodels is a significant cause of transformation change requests [5].

Model transformation patterns include techniques to optimize transformation implementations, particularly to make the evaluation of conditions on models more efficient [11], to enhance the model-processing capabilities of transformations [1], [8], to organize the processing of transformations [26], and to organize the structure of transformation specifications for improved modularity and reuse [12], [14], [26], [29], [43]. There are also patterns specialized for particular kinds of transformation: patterns specific to model-to-text transformation are the use of templates [10] and the use of the Visitor pattern to iterate through a model [13]. A pattern specific to migration transformations is implicit copy by means of retyping source entities to target entities [64].

The classic patterns of [18], [19] are also applicable to model transformations. For example, Visitor and Iterator can be used to define iteration over source model structures, Strategy can be used to select alternative transformation procedures or implementations, whilst Facade can be used to simplify navigation over a complex metamodel, or to encapsulate commonly-used combinations of transformations. In this paper we will only consider patterns specific to model transformations, and not applications of general-purpose patterns.

Transformation patterns involve three main artifacts: the source and target languages, which generally are not modified by application of a pattern, but may be extended to support the internal processing of the transformation (as in Auxiliary Metamodel) or sub-divided (as in Target Model Splitting), and the transformation itself. The structure of the source and/or target languages may determine which patterns are applicable, and how these patterns are applied: for example, a Phased Construction transformation operates based on a particular composition hierarchy of the target language (a partial ordering of target language entity types based upon a notion of one type representing parts of another).

Some transformation languages have expressivity limitations which make it difficult for some patterns to be used with these languages: QVT-R does not have any direct means to constrain the relative order of application of its top-level rules, and this hinders use of patterns based on rule orderings, such as Phased Construction and its specializations. Lack of composition mechanisms for transformations also hinders use of patterns that decompose transformations, such as Parallel Composition and architectural patterns. Some patterns, such as Implicit Copy, require specialized transformation language support, not yet provided by general-purpose transformation languages.

4.1 Classification of model transformation design patterns

Two main categories of transformation design patterns can be identified:

1) **Rule modularization/decomposition patterns** – concerned with restructuring the transformation specification or design to increase its modularity, and to enable the decomposition of a complex transformation into simpler subtransformations, composed e.g., sequentially or in parallel. These are oriented to the improvement of the quality of the specification or design.
   These patterns can be described as restructurings/refactorings of a transformation specification.
2) **Optimization patterns** – concerned with increasing the efficiency of transformation execution, by removing redundant or repeated expression evaluations, optimizing search strategies, or reducing data storage requirements.
   These are implementation-oriented but may be also in some cases described in terms of changes to the transformation specification.

Most of the patterns that we consider in this paper can be grouped into one of these two categories. In addition, some patterns concern special-purpose language facilities, such as model-to-text transformation (Section 7), and others concern increasing the expressiveness of transformation languages (Section 8). Architectural patterns (Section 9) organize systems of transformations in order to improve modularity or processing capabilities.

The following summarizes the classifications of the patterns described in this paper.

1) **Rule modularization patterns:** Phased Construction; Structure Preservation; Entity Splitting/ Structure Elaboration; Entity Merging/Structure Abstraction; Map Objects Before Links; Parallel Composition/Sequential Composition; Auxiliary Metamodel; Construction and Cleanup; Recursive Descent; Replace Explicit Calls by Implicit Calls; Introduce Rule Inheritance.
2) **Optimization patterns:** Unique Instantiation; Object Indexing; Omit Negative Application Conditions; Replace Fixed-point by Bounded Iteration; Decompose Complex Navigations; Restrict Input Ranges; Remove Duplicated Expression Evaluations; Implicit Copy.
3) **Model-to-text patterns:** Model Visitor; Text Templates; Replace Abstract by Concrete Syntax.
4) **Expressiveness patterns:** Simulating Universal Quantification; Simulating Explicit Rule Scheduling.
5) **Architectural patterns:** Phased Model Construction; Target Model Splitting; Model Merging; Auxiliary Models; Filter Before Processing.

In the following sections we give a synopsis of each of these patterns, together with examples of their application in different transformation languages.

5 Rule Modularization Patterns

Rule modularization patterns are important for enabling the practical application of model transformations to large-scale problems, and to improve the structural quality, flexibility, and maintainability of model transformations.

To illustrate these and other patterns we will use a running example based on the well-known UML to relational database refinement transformation [54]. Figure 2 shows the source and target language metamodels of a basic version of this transformation, expressed as instances of the $\mathcal{LMM}$ metamodel, using UML notation (class rectangles for entity types, inheritance arrows for generalization relationships, etc.).

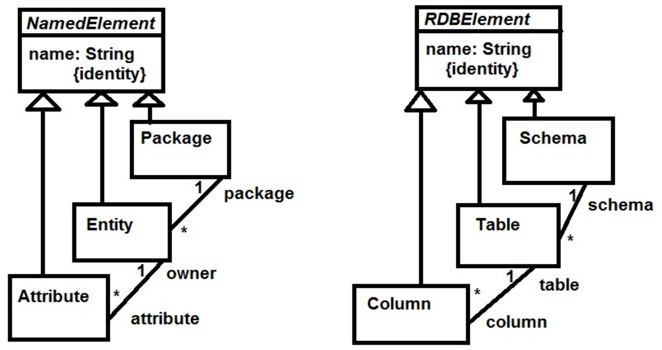

Fig. 2. UML to relational database metamodels

A composition hierarchy of the source language is *Attribute* $<$ *Entity* $<$ *Package*, and *Column* $<$ *Table* $<$ *Schema* for the target language, based on the existence of many-one associations between these entity types.

A specification of the transformation can be written as follows. Two transformation invariants

$$NamedElement \rightarrow isUnique(name)$$
$$RDBElement \rightarrow isUnique(name)$$

declare that no two source objects with duplicate names should exist, and that no two target objects with duplicate names should exist/be created.

The following rule in the generalized rule format

```
for each $p : Package$
create $s : Schema$ satisfying $s.name = p.name$
```

maps packages 1-1 to schemas. For each package p a new schema s is created (unless there is already a schema with the same name, because *RDBElement* :: *name* has been declared as a key attribute). The rule

```
for each $c : Entity$
create $t : Table$ satisfying $t.name = c.name$ and
    $t.schema \simeq c.package$ and
    $c.attribute \rightarrow forAll(a \mid$
        $Column \rightarrow exists(cl \mid$
            $cl.name = a.name$ and $cl : t.column))$
```

maps classes 1-1 to same-named tables, and also populates these tables with columns derived from the attributes of the class. For each attribute a, a corresponding column cl is created (unless there is already a column with the given name, in which case this column is used as cl). The semantics of $E \rightarrow exists(x \mid P)$ is the same as that of **create** $x : E$ **satisfying** P. The table produced from a class is also linked to the schema derived from the package of the class, the notation $t.schema \simeq c.package$ denotes that the $s : Schema$ which has been mapped from *c.package* by a previous application of the first rule should be looked up and assigned as the value of *t.schema*.

Notice that this rule contains navigations in two directions from c and t: up the composition hierarchy to *c.package* and *t.schema*, and down to *c.attribute* and *t.column*.

5.1 Phased Construction

Summary This pattern decomposes a transformation into phases or stages, based on the target model composition structure. These phases can be carried out as separate sub-transformations, composed sequentially. Each phase consists of a rule which creates objects of only one target language entity type, and does not navigate across more than one level of a fixed composition hierarchy of either the source or target languages.

This restricted form of rule structure enables the simplification, analysis and modularization of transformation definitions, and the potential partitioning of large input models into separate parts relevant to each subtransformation.

Application conditions Several warning signs in a transformation specification can indicate that this pattern should be applied: (i) if a transformation rule refers to entities or features across more than two levels of a metamodel composition hierarchy; (ii) if a rule contains, implicitly or explicitly, an alternation of quantifiers $\forall\exists\forall$ or longer alternation chains; (iii) if it involves the creation of more than one target instance.

These are possible signs of a lack of a coherent processing strategy within the transformation, and of excessively complex rules which can hinder comprehension, verification and reuse.

Problems (i), (ii) and (iii) are inter-related: if a rule is attempting to construct target elements at more than one hierarchical level, corresponding to more than one level in the source language, then it will (for example) be navigating from a higher to a lower level *and* referring to the features of lower-level objects, a double navigation. Alternation of quantifiers arise because such rules express "For all $s : Si$, create a $t : Tj$, and for all s' contained in s create a t' contained in t".

Solution The identified rule should be split into separate rules, each relating one source model element (or a group of source model elements) to one target model element, and navigating no further than one step higher or lower in the language composition hierarchies (and not both upwards and downwards).

Figure 3 shows a case of the pattern where a transformation relates part of a source language metamodel (on the LHS) to part of a target language metamodel (on the RHS). Source entity type *Si* is considered higher in the source composition hierarchy structure than *SSub*, and likewise *Ti* in the target language is considered higher than *TSub*. A rule (*rule*2) at the *Si* level maps an *Si* instance satisfying *SCond* to a *Tj* instance satisfying *TCond*, and the rule also navigates down one level to lookup *TSub* elements previously mapped (by *rule*1) from *SSub* elements in order to set the *tr* feature of the *Tj* instances.

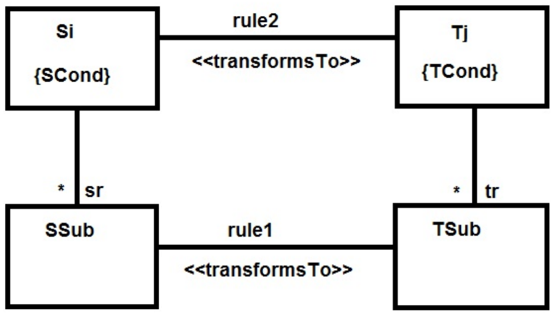

Fig. 3. Phased Construction pattern

Schematically, a rule conforming to this pattern should look like:

for each $s_1 : S_{i1}$; ...; $s_p : S_{ip}$ **satisfying** *SCond*
create $t : T_j$ **satisfying** *TCond and Post*

where the S_{ik} are entities of the source language S, the T_j are entities of the target language T, *SCond* is a predicate on the s_k (identifying which elements the rule should apply to), and *TCond* is a predicate on t, *Post* defines t in terms of the s_k. These predicates may use direct features of the s_k and t, such as *sr* and *tr* in the diagram, but not any longer navigations. There should not be further alternations of quantifiers in *Post*, because such complexity hinders comprehension and analysis. *Post* may be divided into two separate conjuncts, *LPost*, which sets the attribute values of the target object(s), and *GPost*, which sets links between the target object(s) and other objects.

There are two basic variations on the pattern: (i) *Bottom-up* Phased Construction builds a target model starting from entities at the base of the composition hierarchy of the target language, then successively builds composite elements using their part elements; (ii) *Top-down* Phased Construction instead starts from the entities at the top of the target language hierarchy and builds downwards. Figure 3 shows the bottom-up variation.

The choice of one or other variation depends on the structure of the target model. It is generally desirable to maintain the validity of the target model during the transformation (e.g., to simplify verification and composition of the transformation). This means that where there are mandatory associations $T1$ $*$—1 $T2$ in the target model, construction of $T2$ instances should precede construction of $T1$ instances. For $T1$ $*$—$*$ $T2$ associations either order of construction can be chosen, whilst in the case of $T1$ $1..*$—$*$ $T2$ associations, $T1$ instances should be created before $T2$ instances.

For the UML to relational database example, the second rule clearly satisfies the pattern application conditions (i), (ii) and (iii), and we can rewrite the specification into the top-down phased construction form by rewriting the second rule as two new rules:

for each c : *Entity*
create t : *Table* **satisfying** $t.name = c.name$ *and*
$t.schema \simeq c.package$

for each a : *Attribute*
create cl : *Column* **satisfying** $cl.name = a.name$ *and*
$cl.table \simeq a.owner$

This reduces the maximum syntactic complexity of the transformation rules, where syntactic complexity of a rule is defined as the number of occurrences of source or target language element names, plus the number of operators (Section 11): the original rule has 11 feature/entity references, compared to 6 each in the new rules, and 11 operator occurrences compared to 5 each in the new rules. The alternation of quantifiers has been eliminated, and each rule now creates elements of only one entity. The average syntactic complexity of the rules has been reduced from 14.5 to 9.7.

Benefits The pattern improves the modularity of a specification by separating the creation of a composite target entity instance and the creation of its components into separate rules, which also makes the specification more comprehensible. It assists in the scheduling/control of rules, by specifying possible orders for rule execution, based upon the data dependencies of the rules. The pattern enables the decomposition of a transformation into phases, and assists in the definition of an inverse transformation if one exists (see **Applications and examples** below). Verification of the transformation is also facilitated, since it enables decomposition of proofs based on the phases.

Phased Construction potentially enables large input models to be split into two or more smaller models, on the basis of the entity type hierarchy of the source language, and for these models to be separately processed, thus reducing memory requirements. This leads to the Phased Model Construction architectural pattern (Section 9.1).

Disadvantages The number of rules will increase, and some mechanism is needed by which a later rule can look-up elements that were produced by earlier rules, e.g., by some implicit or explicit tracing mechanism, or by Object Indexing (Section 6.2). This may increase the execution cost of the transformation in terms of time and memory.

It is not applicable in cases where condition (ii) holds but the nested forAll-quantifier ranges over a local variable of the rule (such as let-definition variables) as opposed to over a source language feature or composition of features. This is because such local variables cannot be made available to other rules. The third class diagram restructuring rule of [33] is an example of this case.

Applications and examples The pattern is applicable to all categories of transformation except model-to-text (for which case there is no composition hierarchy structure in the target model). It may not be appropriate for refactoring or other update-in-place transformations, since the syntactic non-interference properties (Section 2) may fail for such transformations: their rules may be inherently inter-dependent and so need to be implemented together as parts of a single fixed-point iteration.

A large transformation using the bottom-up version of the Phased Construction pattern is the migration transformation of [40], in UML-RSDS. The UML to relational mapping is also specified using the top-down version of this pattern in Viatra [56], using explicit tracing to look up transformed model elements.

The UML to relational database example of [54] in QVT-R can be expressed using this pattern by using *top relations* instead of invoked relations (relations which are only executed when invoked from the *where* clause of another relation):

```
top relation Package2Schema
{ checkonly domain uml p: Package { name = pn }
  enforce domain rdbms s: Schema { name = pn }
}

top relation Class2Table
{ checkonly domain uml c : Class
  { package = p : Package {}, name = cn }
  enforce domain rdbms t : Table
  { schema = s : Schema {}, name = cn }
  when
  { Package2Schema(p,s); }
  // p already mapped to s
}

top relation Attribute2Column
{ checkonly domain uml a : Attribute
  { owner = c : Class {}, name = an }
  enforce domain rdbms cl : Column
  { table = t : Table {}, name = an }
  when
  { Class2Table(c,t); }
  // c already mapped to t
}
```

Reverse rules for an inverse transformation $\tau^\sim$ can be deduced from transformations τ in phased construction form. These rules express that elements of the target language T can only be created as a result of the application of one of the forward rules: each forward rule R on a single source language entity type S_i:

for each $s : S_i$ **satisfying** *SCond*
create $t : T_j$ **satisfying** *TCond and Post*

has a reverse rule $R^\sim$ with the form:

for each $t : T_j$ **satisfying** *TCond*
create $s : S_i$ **satisfying** *SCond and Post*$^\sim$

where *Post*$^\sim$ expresses the inverse of *Post*. The inverse $\tau^\sim$ of a transformation τ then has rules that are the inverse rules of τ, in the same order. This calculation of $\tau^\sim$ from τ is an example of a higher-order transformation [69].

Related patterns The pattern is closely related to the concept of *phases* in [12] and the concept of *layers* in AGG [68]. In UML-RSDS this pattern is termed *Conjunctive-implicative form* at the specification level, and *Phased creation* at the design level [43].

One motivation for applying the Phased Construction pattern is that it provides improved modularity compared to versions of a transformation without the pattern. In particular, many transformation languages such as QVT-R, ATL and ETL adopt a 'recursive descent' style of specification, in which top-level rules apply to the entities at the top of a composition hierarchy in a source language metamodel (such as Packages in the UML to relational database transformation of Figure 2) and these then invoke subordinate rules at successively lower levels of the source language composition hierarchy.

Analysis of the semantics of QVT-R transformations [9] has shown that these typically follow the recursive descent specification style, with top relations invoking non-top relations. Each such invocation may introduce a further implicit alternation of quantifiers. Our observation is that a Phased Construction approach would be a preferable alternative in many cases.

5.2 Structure Preservation

Summary This is the simplest special case of the Phased Construction pattern where the entity types of S are in 1-1 relation to those of T, with the objects of each pair S_i and T_i of entities also in a 1-1 relation. Such transformations occur as copying transformations in ATL and other languages.

Application conditions This pattern is appropriate if the transformation should maintain a 1-1 relation between the source and target models. In particular, the pattern is used when a model needs to be copied to a closely-related language. These transformations can be used as simple sub-components within more complex transformations, such as migration transformations where only a small part of a model changes significantly in structure from source to target.

Solution Write the transformation rules in the phased construction form

for each $s : S_i$ **satisfying** *SCond*
create $t : T_i$ **satisfying** *TCond and Post*

where *Post* simply copies the feature values of s to corresponding features of t. If identities of S_i objects are copied to those of T_i objects, this mapping must be in an injective (inequality-preserving) manner. Either top-down or bottom-up construction strategies can be used, as with Phased Construction.

Applications and examples The pattern is applicable to the same categories of transformations as Phased Construction, but is particularly relevant for migration transformations.

Transformations which copy models are examples of this pattern. Our example UML to relational database transformation is essentially a structure preservation pattern because of the structural isomorphism between the UML and relational database languages in Figure 2.

The inverse of a structure preservation transformation is itself typically a structure preservation transformation.

Related patterns The pattern is a special case of Phased Construction (Section 5.1). The pattern is referred to as the *mapping* pattern in [26]. In [20] a pattern for implementing copy transformations in QVT-R using marking relations is described, and [69] considers the generic specification of copy transformations in ATL. Structure Preservation can be used as a basis for implementing update-in-place transformations by selective copy, for languages which do not directly support update-in-place processing. The ATL and QVT-R solutions in [33] are examples of this technique.

The Implicit Copy (Section 6.8) pattern is another solution to the problem of defining transformations that preserve large parts of a model structure, but this pattern requires suitable language facilities to be present in the transformation language [64].

5.3 Entity Splitting

Summary A transformation contains rules which (i) create instances of two or more target entity types from each instance of a source entity types, and where these instances are not part of a single conceptual unit; or which (ii) maps instances of one source entity type to instances of different alternative target entity types, depending on disjoint conditions.

In either case separate rules should be used to define the creation of the separate/alternative target instances.

Application conditions If a transformation rule combines the creation of multiple target instances in a single rule, the rule may become excessively complex and difficult to maintain. This problem is an example of *tangling* of functionality in a single rule, and of lack of *cohesion* in the rule effect.

Solution Separate the parts of the rule that create/update separate target instances into separate rules. If one entity type $T2$ depends on another, $T1$, then the rule creating $T1$ should precede the rule creating $T2$, otherwise they can occur in either order. Identity attributes or traces may be used to relate the different target instances derived from a single source instance. These attributes/traces are also necessary for correct reversal of the transformation.

Figure 4 shows the typical structure of this pattern, as an object diagram relating one source object $s : S1$ to multiple target objects $t1 : T1$, $t2 : T2$, $t3 : T3$ via separate rules.

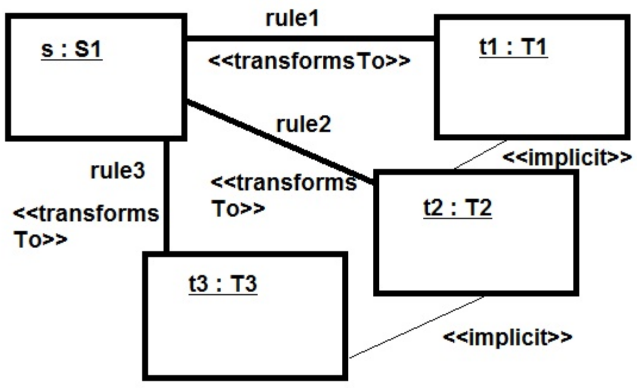

Fig. 4. Entity Splitting pattern

If the instances created actually form closely bound parts of a single concept (e.g., if classes are mapped to both a table and its primary key, in a UML to relational database mapping), then the rule should not be split. This variation of the pattern could be termed *Structure Elaboration*. One way of identifying this case is that the rule establishes links of an aggregation association or other mutually mandatory relationship between the respective target elements, such as $T1$ 1..∗—1 $T2$ or $T1$ 1—1 $T2$ associations.

Benefits This pattern assists in improving the modularity and cohesion of a specification, separating into distinct rules the construction of distinct target entities. The opportunities for parallel execution of rules are increased.

Disadvantages The number of rules is increased. The efficiency of the transformation may be decreased, since there are now multiple iterations over the source entity type (e.g., $S1$), instead of a single iteration.

Applications and examples The pattern is applicable to the same categories of transformations as Phased Construction, but is particularly relevant for refinement transformations.

An example is given in [35] of the application of this pattern to the creation of MVC components from source data: source model objects of types such as *OpenElement* are transformed in a single rule into interconnected model, view and controller objects *Open*, *OpenView*, *OpenController* (Figure 5).

The single rule for *OpenElement* could be written as:

```
for each $e : OpenElement$
create $t : OpenView;\ op : Open;\ oc : OpenController$
satisfying $t.obsId = e.id$ and
    $op.modId = e.id$ and $op.observer = t$ and
    $oc.view = t$
```

Since the entities *Open* and *OpenController* depend on (refer to) *OpenView*, we can split the rule into three separate rules using the pattern: the first creates *OpenView* objects, the second creates *Open* objects and links each of these to the *OpenView* objects with the same id value, and the third does the same for *OpenController*.

The motivation for applying the Entity Splitting pattern in this case is to avoid the tangling of several different source-target relationships in a single rule, and instead to separate out these relationships into different rules. It may be necessary to define in the separate target entities some key attribute or stereotype which records the fact of the semantic link between them (i.e., that they represent separate parts of the same source entity). This allows the definition of a semantic interpretation and of a reverse relation.

For example, consider the simple case where there is one source entity with attributes $att1$, $att2$, and two target entities each with one of the attributes:

```
for each $s : S_1$
create $t1 : T_1;\ t2 : T_2$
satisfying $t1.att3 = s.att1$ and $t2.att4 = s.att2$
```

This can be split into two simpler rules which each refer only to one target entity type:

```
for each $s : S_1$
create $t1 : T_1$ satisfying $t1.att3 = s.att1$

for each $s : S_1$
create $t2 : T_2$ satisfying $t2.att4 = s.att2$
```

Notice that rule complexity *comp* (Section 11) has been decreased in this case: from 13 for the original rule to 7 for each of the new rules.

Other examples are refinement or code-generation transformations such as the generation of multiple Java Enterprise Edition elements (database tables, value objects, EJBs) from single UML classes.

Related patterns The pattern is a special case of Phased Construction (Section 5.1). The pattern is termed *local source to global target* in [12]. The *refinement* pattern in [26] corresponds to Structure Elaboration. The *element mapping* and *element mapping with variability* patterns of [29] are special cases of the pattern. The combination of two rules using the fork operator of [59] is in a sense an inverse application of this pattern.

The Target Model Splitting architectural pattern (Section 9.2) can be used to store in separate models the instances of separate target entity types T_1 and T_2 derived from a source entity S_1, if there are only implicit links between the two target types, and not explicit object references (as in Figure 5).

5.4 Entity Merging

Summary One target model entity type is updated using data from two or more different source model entity types. Separate rules should be used to define the updates from separate sources.

Application conditions This pattern is used when a target entity type amalgamates data from several different source entities. The target entity type will appear in the succedent of two or more rules and each rule will create or update instances of it.

Solution Define the separate updates as distinct rules, creating the target instance in the first rule, and using some means of object lookup to locate this instance for update by subsequent rules.

Figure 6 shows a typical structure of this pattern as an object diagram.

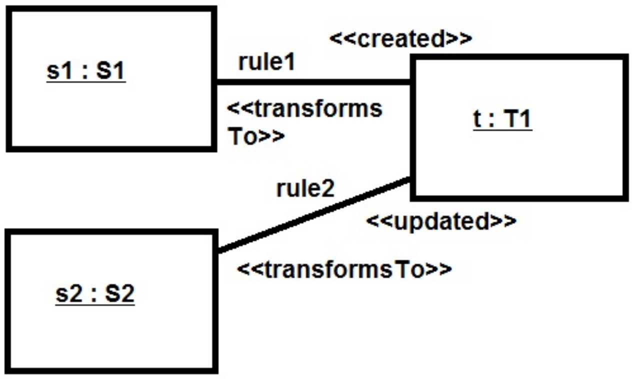

Fig. 6. Entity Merging pattern

In cases where a closely semantically related group of source elements are mapped to a single target element,

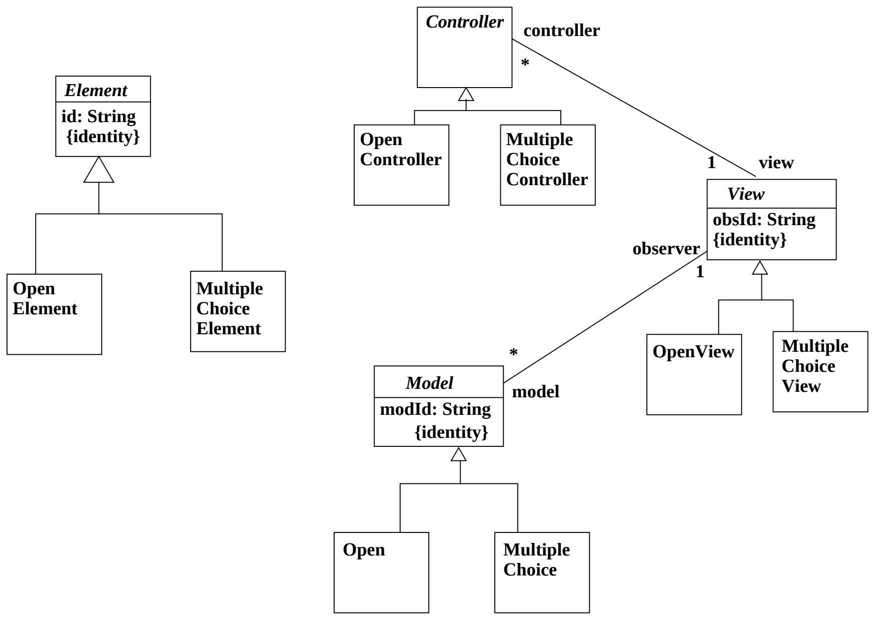

Fig. 5. MVC introduction metamodels

using a multiple matching rule, separate rules are not suitable. This could be termed the *Structure Abstraction* pattern.

Benefits This pattern can be used to improve the decomposition of the specification, simplifying individual rules. It should be checked that rules R_i and R_j which may update the same instance do not semantically interfere with each other: they either update entirely separate and independent sets of features of the instance, or they update shared features (which must be set-valued association ends), in a consistent and order-independent way (by both adding elements to the association ends). This enables the rules to be implemented by successive phases.

The mapping of identity attributes from source objects to target objects can be non-injective, but in such a case successive updates to the same target object from different source objects (as in Figure 6) must also be non-interfering.

Applications and examples The pattern is applicable to the same categories of transformations as Phased Construction, but it is particularly relevant to abstraction transformations or to model-merging transformations, which combine two separate models into a single model.

An alternative form of the UML to relational database transformation takes account of inheritance in the source model, and only creates relational tables for root classes in the source model inheritance hierarchy, the data of subclasses is then merged into these tables (Figure 7).

```
for each $e : Entity$ satisfying $e.parent = Set\{\}$
create $t : Table$ satisfying $t.name = e.name$
```

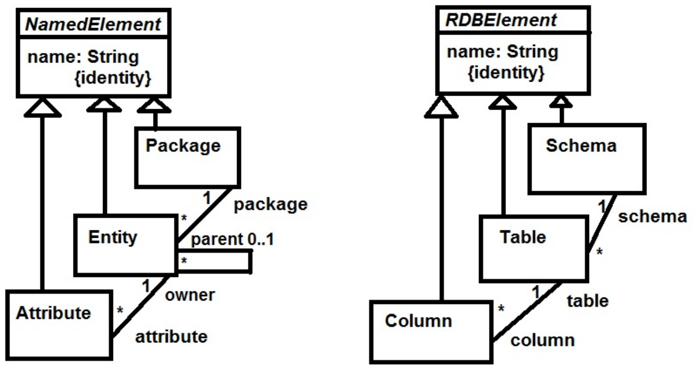

Fig. 7. UML to relational database metamodels (enhanced)

```
for each $c : Entity$; $a : c.attribute$
create $cl : Column$
satisfying $cl.name = a.name$ and
    $cl : Table[c.rootClass().name].column$
```

where *rootClass*() is a recursively-defined operation of *Entity* which returns the root class of the entity. The notation $E[v]$ denotes the element of entity E with primary key value v (Section 6.2).

Tables are created (for root classes only) by the first rule and then updated by the second, so this transformation conforms to the pattern.

Related patterns The pattern is a special case of Phased Construction (Section 5.1). This pattern is referred to as the *global source to local target* type of transformation in [12]. The *node abstraction* pattern in [26] corresponds to a case of Structure Abstraction, as does *flattening*. The reverse transformation of an entity merging is an entity splitting transformation.

In order to look up already existing $t : T1$ objects, to update them with additional data, some mechanism such as trace-based lookup or the Object Indexing pattern (Section 6.2) must be used.

The pattern can be extended to the Model Merging architectural pattern (Section 9.3) in cases where the input model can be split into smaller models, together with a separation of the transformation into target creation and elaboration sub-transformations.

5.5 Parallel Composition

Summary The construction of a transformation as a composition of components which define separate aspects of the transformation task, and which can be performed in parallel or sequentially.

Application conditions This pattern is applicable whenever a transformation consists of at least two distinguishable groups of rules which can be separated to reduce the complexity and to increase the modularity and cohesion of the transformation.

More precisely, let $wr(R)$ denote the set of entity types and features updated by a rule R, and $rd(R)$ the set of entity types and features that are read by R. A *slice* $\tau \lhd V$ of a transformation τ with respect to set V of entity types and features is the subtransformation of τ containing only those rules R with $wr(R) \subseteq V$.

Then τ can be decomposed into parallel subtransformations

$$\tau_1 = \tau \lhd V1$$
$$\tau_2 = \tau \lhd V2$$

if there are disjoint sets $V1$, $V2$ which together partition $wr(\tau)$ and such that the sets of rules of τ_1 and τ_2 partition the set of rules of τ and

$$wr(\tau_1) \cap rd(\tau_2) = \{\}$$
$$wr(\tau_2) \cap rd(\tau_1) = \{\}$$

Solution Separate the specification rules into distinct groups based on the aspect of the target model which they affect. Typically the groups are defined to update disjoint sets of target entity types and features. The groups can then be factored into separate transformations.

If only the condition $wr(\tau_2) \cap rd(\tau_1) = \{\}$ holds, then τ can be decomposed sequentially as $\tau_1;\ \tau_2$, the special case of a *Sequential Composition* pattern. In general, a loop-free activity can be derived from the transformation specification: the activity nodes are the slices of the specification, and are composed using sequencing or parallel composition depending on their relative dependencies. The activity implements the original specification.

Benefits This pattern improves the specification modularity by separating out related rules into subtransformations, which may be applied and verified and modified relatively independently. This decomposition potentially improves the reusability and flexibility of the specification: individual components can be modified because of metamodel evolution or requirements changes, relatively independently of other components. Conceptually it simplifies the specification by separating out distinct 'aspects' of the transformation. The cohesion measure $cohe(\tau)$ of a specification (Section 11) is improved for the parallel sub-transformations compared to the original transformation.

The pattern can be used for update-in-place transformations, in order to minimize the number of rules which need to be implemented using a fixed-point technique. For example, a transformation τ which overall is update-in-place could be decomposed into a composition of an input-preserving transformation τ_1 and an update-in-place transformation τ_2, as in the example of Section 13.

Disadvantages The groups of rules selected should be cohesive, and with few dependencies upon other groups. Cycles of dependencies between groups are not permitted.

Applications and examples The pattern is applicable to all categories of transformations.

The example of [35] can be further decomposed using this pattern into three separate transformations: the first transformation *sourceToViews* includes the rules defining the mapping of elements to view objects. This is sequentially followed by a parallel composition of transformations *sourceToModels* containing the rules creating model objects, and *sourceToControllers* containing the rules defining controllers.

The *sourceToViews* transformation creates views for *Open* and *MultipleChoice* elements:

```
for each $e : OpenElement$
create $t : OpenView$ satisfying $t.obsId = e.id$

for each $e : MultipleChoiceElement$
create $t : MultipleChoiceView$ satisfying $t.obsId = e.id$
```

The *sourceToModels* transformation creates models for each kind of element and links these to corresponding views:

```
for each $e : OpenElement$
create $t : Open$ satisfying
    $t.modId = e.id$ and $t.observer = OpenView[e.id]$

for each $e : MultipleChoiceElement$
create $t : MultipleChoice$ satisfying
    $t.modId = e.id$ and
    $t.observer = MultipleChoiceView[e.id]$
```

The *sourceToControllers* transformation creates controllers for each kind of element and links these to corresponding views:

```
for each $e : OpenElement$
create $t : OpenController$ satisfying
    $t.view = OpenView[e.id]$
```

```
for each $e : MultipleChoiceElement$
create $t : MultipleChoiceController$ satisfying
    $t.view = MultipleChoiceView[e.id]$
```

Likewise, a UML to Java Enterprise Edition transformation could be decomposed into subtransformations which map the UML model to separate Java EE tiers.

Related patterns The pattern is referred to as *aspect-driven transformation* in [14]. The pattern can be used as a further evolution of the Entity Splitting (Section 5.3) pattern to decompose transformations that map one source entity type to multiple target entity types.

The external composition of transformations by the fork operator of [59] is similar to parallel composition of transformations. In the UML-RSDS approach this pattern is implemented by means of the *extend* relation between use cases: the separate transformations are defined as use cases related by extension. Their extend-composition is equivalent to the original transformation [47].

5.6 Map Objects Before Links

Summary To define a correct and efficient transformation specification when there are cycles of entity type dependencies in the source model.

Application conditions Applicable whenever the source model contains self-associations on entities or longer cycles of entity type dependencies, which need to be mapped to similar target structures by the transformation. A strictly hierarchical phased construction form is not suitable in this case.

Solution Split the transformation specification into two phases, the first maps source entity types and their attributes and any other data which is not involved in dependency cycles. The second phase links together the target objects based on the source model links, using traces or identity attributes to look up the target objects that should be linked.

Benefits The specification is decomposed into phases, and the complexity of individual rules should be reduced. Bounded iteration can be used to implement rules instead of fixed-point iteration.

Disadvantages The specification of local postconditions *LPost* (setting attribute features) and global postconditions *GPost* (setting association end features) for single source entity types becomes split into separate rules, an example of *scattering* of specification text. In addition, more iterations are potentially required over the source language entity types.

For transformations where contextual information needs to be passed from the processing of composite objects to the processing of their components, Recursive Descent is more appropriate.

Applications and examples The pattern is applicable to the same categories of transformations as Phased Construction.

Figure 8 shows a typical example of a situation where the pattern is relevant, where in the source metamodel, entity E depends upon itself via the subclass association role *elements*. The target metamodel has a similar recursive structure based on entity F.

Applying the bottom-up Phased Construction pattern directly to this problem (with $D < EBasic$, $D < EComp$, $E < EComp$, $G < FBasic$, $G < FComp$, $F < FComp$) results in a specification of the form:

```
for each $d : D$
create $g : G$ satisfying $PostDG$

for each $e : EBasic$
create $f : FBasic$ satisfying
        $LPostEBasicFBasic$ and $f.gr \simeq e.dr$

for each $e : EComp$
create $f : FComp$ satisfying
        $LPostECompFComp$ and
        $f.gr \simeq e.dr$ and $f.elements \simeq e.elements$
```

where D objects are mapped to G objects in the first rule, and *EBasic* objects to *FBasic* objects with corresponding G objects in the second.

In the final rule *EComp* objects are mapped to *FComp* objects with corresponding *elements*. However there is a problem with this rule in that F is both written and read (in the lookup of the F objects corresponding to *e.elements*), so the rule needs a fixed-point iteration implementation. The problem is that not all of *e.elements* may have been mapped to F elements when the mapping of e is attempted – so the mapping of e would need to be redone whenever one of its *elements* is mapped. This approach is therefore quite inefficient.

By applying the Map Objects Before Links pattern, we can instead separate the object mapping and link mapping. Three rules map D to G, *EBasic* to *FBasic* and *EComp* to *FComp* in a first phase, without establishing any links between the target objects:

```
for each $d : D$
create $g : G$ satisfying $PostDG$

for each $e : EBasic$
create $f : FBasic$ satisfying $LPostEBasicFBasic$

for each $e : EComp$
create $f : FComp$ satisfying $LPostECompFComp$
```

In a second phase the links from F to G objects, and from *FComp* to F objects are established, based on corresponding links in the source model:

```
for each $e : E$
do $F[e.id].gr = G[e.dr.id]$

for each $e : EComp$
do $FComp[e.id].elements = F[e.elements.id]$
```

The pattern of this second phase is characteristic of Map Objects Before Links, and inverse rules of the same form

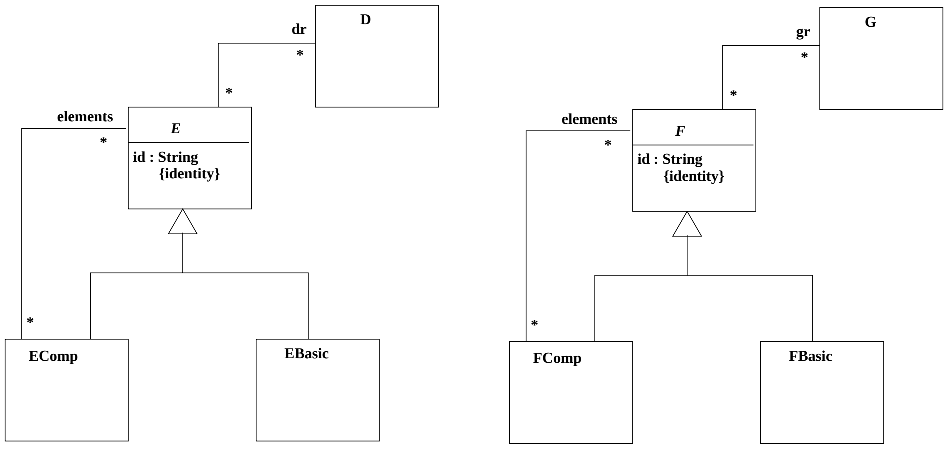

Fig. 8. Example of cyclic dependency

can be directly derived by interchanging the source and target language elements:

for each $f : F$
do $E[f.id].dr = D[f.gr.id]$

for each $f : FComp$
do $EComp[f.id].elements = E[f.elements.id]$

Another example of this situation is the GMF migration transformation of [42], the *initialize* transformation of Section 13, and the mapping of testcase-based specifications to designs [44].

A variation on the pattern is where new instances in the target model need to be created for each link in the source (as opposed to simply linking target instances for each source link). The Tree to Graph transformation of [34] is an example of this variation. In the source language there is a *-to-0..1 self-association *parent* from *Tree* to itself. The original ETL specification uses trace lookup and implicit rule invocation:

```
rule Tree2Node
  transform t : Tree!Tree
  to n : Graph!Node
  { n.label := t.label;
    if (t.parent.isDefined())
    { var edge := new Graph!Edge;
      edge.source := n;
      edge.target := t.parent.equivalent();
    }
  }
```

The *obj.equivalent*() expression looks up in the transformation trace to check if *obj* has already been mapped to a target element *tobj*, if so, it returns such an element, otherwise it invokes any applicable rules (in this case, *Tree*2*Node* itself) to map *obj* to a target element, which is then returned. This solution therefore uses a form of the Recursive Descent pattern, Section 5.9, to process the source language self-association *parent* (processing of a tree t may lead to processing of $t.parent$, then $t.parent.parent$, etc.).

Applying the Map Objects Before Links pattern, a first rule creates a node for each tree:

for each $t : Tree$
create $n : Node$ **satisfying** $n.label = t.label$

A second rule then creates edges for each link between parent and child trees:

for each $t : Tree$; $p : Tree$ **satisfying** $p : t.parent$
create $e : Edge$ **satisfying**
$e.source = Node[t.label]$ *and* $e.target = Node[p.label]$

This solution avoids implicit invocation, and has a stronger control over the ordering of instance creation. Bounded iterations can be used to implement each rule so that its time complexity and termination are simpler to establish.

The default execution mode of ATL uses this pattern: target objects are created from source objects in an initial phase, then inter-related by a second phase [15].

Related patterns The pattern is a special case of Phased Construction (Section 5.1) which deals with self-associations and other circular data relations in the source model. It is described as the separation of *generate* and *refinement* transformation phases in [12]. The pattern uses Entity Merging because target objects are typically created in an initial phase and updated with links in a second phase.

5.7 Auxiliary Metamodel

Summary The introduction of a metamodel for auxiliary data or operations, neither part of the source or target language, used in a model transformation.

Application conditions Signs that this pattern is necessary include: (i) excessively complex expressions read within rules; (ii) duplicated expressions occur across different rules; (iii) complex ad-hoc data structures are used within the transformation.

Problem (i) indicates that auxiliary data needs to be defined to store the expression values or to simplify their computation. A typical case is a query transformation which evaluates some complex expression over the source model, such as counting the number of instances of a complex structure in the source model: explicitly representing the computed data by auxiliary entity types or features may simplify the transformation. Problem (ii) indicates that the duplicated expressions should be precomputed and stored in some auxiliary data, or computed by auxiliary operations, to avoid duplicated evaluations. Problem (iii) indicates that appropriate new entity types (not present in either source or target languages) need to be defined to hold the data structures that the transformation uses during its processing.

Solution Define the auxiliary metamodel as a set of (meta) attributes, associations, entity types, operations and generalizations additional to the source and/or target metamodels. The auxiliary data may be used in the succedents of rules (to define how the auxiliary data is derived from source model data) or in antecedents (to define how target model data is derived from the auxiliary data).

Figure 9 shows a typical structure of the pattern in terms of the language metamodels. The auxiliary metamodel simplifies the mapping between source and target by factoring it into two steps, or it can be used to store information about the mapping as it proceeds. All lines between entity types represent ≪ *transformsTo* ≫ relationships.

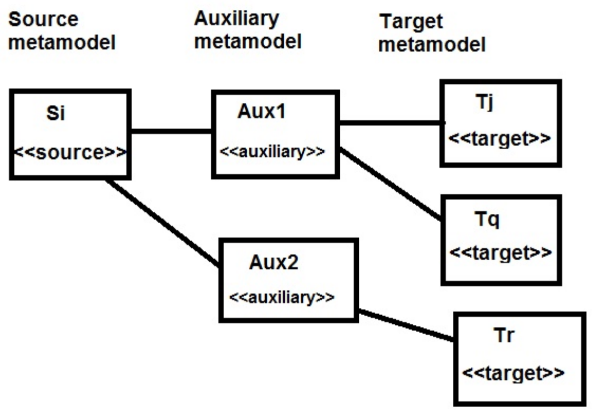

Fig. 9. Auxiliary Metamodel structure

Benefits This pattern helps to simplify the complexity of model navigations and constructions in a transformation, and to decompose the transformation into subparts/phases. In particular, complex relationships between a source and target model can be decomposed into simpler relationships between these models and an intermediate model using the auxiliary metamodel.

Disadvantages It may be necessary to remove auxiliary data from a target model, if this model must conform to a specific target language at termination of the transformation. A final phase in the transformation could be defined to delete the data (cf. the Construction and Cleanup pattern, Section 5.8).

Applications and examples This pattern is applicable to all categories of transformations. It is a strong candidate for inclusion as an in-built facility in model transformation languages, because of its wide applicability. UML-RSDS provides support for the pattern by allowing new entity types stereotyped as ≪ *auxiliary* ≫ to be introduced in a transformation specification [47]. Likewise in graph transformation languages such as TGG [71], *correspondence graphs* can be defined using auxiliary classes and associations. These record detailed traces to assist in the control of the transformation.

An example where complex data structures used during a transformation could be simplified by using Auxiliary Metamodel is the Kermeta solution of [33]. Sets of sets of *Property* instances are created and manipulated internally in this transformation. Likewise, complex tuple structures are constructed in the ATL solution to the class diagram to relational database problem (www.eclipse.org/atl/atlTransformations/SimpleClass2SimpleRDBMS/).

Auxiliary Metamodel can be used to add artificial structure to a model, such as a root element, to assist in navigation. It can be used to store transformation parameter data, to enable parameterization of a transformation. It can also be used to precompute expression values prior to a transformation execution, to avoid duplicated evaluations, or to simulate multiple element matching in rules by single element matching [8]. For example, if the instances of a source entity E need to be sorted in a particular order, for processing by several rules, an auxiliary collection can be introduced to hold the sorted version of $E.allInstances()$.

In the state machine slicing transformation of [39], additional associations recording dependency sets of variables in each state, and the reachability relation between states, need to be added to the core state machine metamodel. Likewise for the lambda-calculus restructuring transformation (Section 12 and [21]), an additional association recording the set of bound variables in scope in each expression is required. Auxiliary metamodels are also used to implement explicit tracing facilities. In transformation languages such as Viatra [56] and Kermeta [32], the auxiliary entities and associations record information such as a history of rules applied and connections between target model elements and the source model elements they were derived from. In the UML to relational database example, there could be auxiliary trace classes *Package2Schema*, *Entity2Table* and *Attribute2Column* linking respective source and target elements:

```
for each $c : Entity$
create $t : Table; c2t : Entity2Table$
satisfying $t.name = c.name$ and
        $t.schema \simeq c.package$ and
        $c2t.source = c$ and $c2t.target = t$
```

```
**for each** $a$ : *Attribute*
**create** $cl$ : *Column*; $a2c$ : *Attribute2Column*
**satisfying** $cl.name = a.name$ and
        $cl.table \simeq a.owner$ and
        $a2c.source = a$ and $a2c.target = cl$
```

Traces (for 1-many relationships from source to target) can also be recorded by introducing auxiliary identity attributes into source and target entity types, with the values of these being used to identify which target instances correspond to (have been derived from) which source instances.

Related patterns Special cases of this pattern are the *map-using-link* pattern of [1], the *determining an opposite relationship* and *finding constant expressions* patterns of [11], and the *transformation parameters* pattern of [8]. The *multiple matching* pattern of [8] can be implemented using an auxiliary metamodel entity to store single objects representing groups of elements that together satisfy some matching criterion. The classic programming technique of *memoisation* [52] also introduces auxiliary data structures in order to avoid redundant computations.

The pattern can evolve to the Auxiliary Models pattern, Section 9.4, if separate models of auxiliary or intermediate data are required to carry out the transformation.

5.8 Construction and Cleanup

Summary This pattern structures a transformation by separating rules which construct model elements from those which delete elements.

Application conditions Potentially applicable if a transformation must remove some elements of a model, in addition to constructing elements. It is particularly relevant if redundant objects may be created during the transformation processing, which, after a certain point in the processing, are no longer required and should be deleted to reduce memory requirements.

Solution Separate the creation phase and deletion phase into separate rules, usually the creation (construction phase) will precede the deletion (cleanup). These can be implemented as separate transformations, each with a simpler specification and coding than the single transformation. Alternatively, deletion rules may be introduced at intermediate points during the transformation to remove objects which are not subsequently needed.

If entity types A and B are related by an association with one mandatory end, e.g., $A *\!\!-\!\!\!-1\ B$, then instances of the optional end entity should be deleted before instances of the mandatory end. If both ends are mandatory, then linked instances should be deleted in the same rule.

Benefits This pattern assists in modularizing a transformation. It potentially reduces memory requirements by removing unnecessary objects.

Disadvantages The pattern leads to the production of intermediate models (between construction and deletion) which may be invalid as models of either the source or target languages. It may be necessary to form an enlarged language for such models. Cleanup transformations, like filters (Section 9.5) may involve fixed-point processing because of cascaded deletion effects.

Applications and examples Examples are migration transformations where there are common entities between the source and target languages [42]. A first phase copies/adapts any necessary data from the old version (source) entities which are absent in the new version (target) language, and creates data for new entities. Then a second phase removes all elements of the model which are not in the target language. The intermediate model is a model of a union language of the source and target languages.

Other examples are complex update-in-place transformations, such as the removal of duplicated attributes [33] or multiple inheritance. The ATL solution to the case study of [33] uses this pattern, the cleanup rule:

```
rule DeleteDuplicatedProperty{
from p : ClassDiagram!Property(
  ClassDiagram!Generalization->
    allInstances()->exists(g |
      g.specific.OwnedAttribute->includes(p)
      and
      g.general.OwnedAttribute->exists(pr |
             pr.name = p.name and
             pr.type = p.type)))
  to
      drop
}
```

is executed after each rule which creates a superclass attribute that abstracts duplicated subclass attributes [33]. The cleanup rule removes all attributes p from direct subclasses of a class which have the same name and type as an attribute pr of the class.

Related patterns The classic programming concept of garbage collection is analogous to this pattern. The pattern is dual to pre-filtering of input models by Filter Before Processing (Section 9.5).

5.9 Recursive Descent

Summary Construct target model elements recursively from the top down, by following the corresponding hierarchy of the source model elements from composite elements down to their components, recursively.

Application conditions Appropriate for transformations based on recursive source model structures, such as abstract syntax trees of expressions, or other cases where source model elements have an internal hierarchical structure. The pattern is also applicable if individual applications of one rule $R1$ must be sequentially chained with applications of another rule $R2$.

Solution Decompose the transformation into called operations, based upon the hierarchical structure of source model elements. Processing is initiated on elements at the topmost level of structure. This processing then invokes processing of the successive levels of internal structure of these elements.

A rule

> **for each** $s : S_i$ **satisfying** *SCond*
> **create** $t : T_j$ **satisfying** *TCond and Post*

where S_i and T_j are maximal in the entity hierarchies of their languages will be implemented by an operation $mapSiToTj() : T_j$ of S_i, which if the S_i object satisfies *SCond*, creates a new instance t of T_j, sets its local attribute data according to the local postcondition parts *TCond* and *LPost*, and recursively calls $subs.mapSubSToSubT(t)$ on subordinate elements *subs* of the S_i object which are involved in the transformation. Rules for the subordinate elements are likewise implemented by operations $mapSubSToSubT(t : TSup)$ with the same structure, but which also set the links specified in the global postcondition part *GPost* between t and its subordinate components.

To implement rule chaining of rule $R1$ followed by $R2$, the succedent of $R1$ explicitly invokes the operation representing rule $R2$ as its final action.

Benefits The pattern decomposes a transformation into a set of operation definitions, and specifies an explicit scheduling of rule applications. It may be more efficient than Phased Construction if only parts of the source model need to be examined or mapped, since the recursive descent will only inspect elements that are relevant to the transformation, rather than all source model elements.

It enables contextual information from the processing of container objects to be passed down to the processing of their components.

Disadvantages The navigation defined by the pattern is tightly coupled to the source and target entity hierarchies, which is a similar problem to the classic Visitor pattern. The use of explicit rule invocation by operation calls reduces the flexibility of the transformation. *LPost* and *GPost* for the same source and target instances are split into separate operations, an example of scattering of the specification. Verification requires the use of induction over the recursive calls.

Phased Construction will generally be more efficient for cases where the input model does not have recursive structures: in this case a suitable ordering of rules can ensure that elements are always created by prior rules before they are looked-up by subsequent rules, so avoiding the cost of testing for the existence of such elements.

Applications and examples This pattern is applicable to all categories of transformation.

The implementation of the UML to relational database transformation in QVT-R, ATL or KMTL are examples of this pattern [2], [54], and this is the usual specification approach adopted in QVT-R, with subordinate rules being explicitly invoked in the *where* clause of other rules. In ATL, explicit invocation of called rules can be used. The mapping of one form of abstract syntax representation of λ-expressions to another form (Section 12) is also naturally solved using this pattern, because contextual information needs to be passed down from enclosing expressions to sub-expressions when processing the expression structures.

An example of rule chaining is from the GrGen.NET solution to the class diagram rationalization problem [33]:

```
rule rule1 {
  c : Class;
  ...
  }
  modify
    c -:ownedAttribute-> a4 : Property
      -:type-> t;
    eval {
      a4._name = a1._name;
    }
    exec(RemoveAttributeFromSubclasses(c, a4)
      ;> [createInverseEdges]);
  }
}
```

In the *exec* clause, applications of *RemoveAttributeFromSubclasses* and *createInverseEdges* are explicitly chained after the main rule by means of invocation.

Related patterns Recursive descent is a general strategy for processing hierarchically-structured data, e.g., standard algorithms for tree data structures.

This pattern is an alternative to Map Objects Before Links (Section 5.6), for migration/refinement/abstraction of recursively structured source models, and is preferred if the transformation of source elements depends upon contextual information from these enclosing structures.

Rule chaining can alternatively be specified using forms of higher-order parameterization [72], where a successor rule is defined generically as a parameter of a predecessor rule, to avoid explicit named dependence between the rules. This requires appropriate language support.

The UML-RSDS toolset provides a meta-transformation to convert recursive descent style specifications into phased construction style specifications, by introducing an auxiliary trace association (Appendix).

5.10 Replace Explicit Rule Invocation by Implicit

Summary Explicit calls of one rule within another can make it difficult to evolve transformations. This pattern replaces such calls by implicit calls.

Application conditions If a rule operating on source elements at one level of composition structure explicitly invokes rules at a lower (or higher) level, then these invocations could be replaced by implicit calls if the transformation language supports this.

Solution For inter-rule invocation, we can distinguish *implicit* invocation, where the caller does not explicitly name or identify the called rule, and *explicit* invocation, where the called rule is explicitly identified by the caller.

If rule $R1$ maps elements $s : S1$ to elements $t : T1$, it will typically need to assemble elements of any entity $T2$

subordinate to t, which correspond to source elements $s2 : S2$ subordinate to s. This can be done by explicit invocation of rules $R2$, $R3$ etc., to map the subelements (as in the Recursive Descent pattern, Section 5.9), or by implicit invocation or matching of these subelements (e.g., using a mechanism such as *equivalent*/*equivalents* in ETL).

Benefits The implicit call approach has the advantage that changes to the way that subordinate elements are mapped does not affect the caller, since the caller is unaware of which rules are used to carry out these mappings. Therefore, the part of the transformation that manages the $S2$ to $T2$ mapping can be modified in its structure and details without affecting the syntax of the $S1$ to $T1$ mapping.

Disadvantages The interprocedural dependencies in the transformation are disguised by implicit calls. In reasoning about the rules, we need to analyse all possible cases of rules that may be invoked by the implicit call. There may be a small loss of efficiency in implementing implicit calls, due to the need to resolve the actual target of the call.

Applications and examples This pattern is applicable to all categories of transformation.

ETL has an implicit invocation mechanism, which is used when a rule needs to convert source model elements *s.ssub* of entity $S2$ subordinate to one of its source elements $s : S1$ to corresponding elements of a target entity $T2$, in order to assemble the subelements *t.tsub* of the image $t : T1$ of s. A rule can refer to *s.ssub.equivalents*() to recover the $T2$ elements that have been mapped from the *s.ssub* elements, or to execute rules to create the $T2$ elements if they do not already exist [34]. This is in contrast to explicit invocation, for example in QVT-R, where the conversion is carried out by calling an explicitly named rule in the *where* clause of the calling rule.

Other transformation languages also support implicit calls or equivalent mechanisms. In UML-RSDS, a constraint mapping an element *self* : $S1$ with a feature *self.ssub* : *Set*($S2$) to an element $t : T1$ with feature *t.tsub* : *Set*($T2$) will look up the $T2$ elements corresponding to *self.ssub*, using identity attributes, and this lookup is independent of the construction process used to derive these elements:

```
T1->exists( t  | t.tsub = T2[ssub.id] )
```

However, this does not create the subordinate elements if they do not already exist. Instead, the constraints must be ordered so that all constraints that create $T2$ elements precede constraints that use them.

To imitate the ETL implicit invocation technique, the rule would need to be written as:

```
T1->exists( t | ssub->forAll( s2 |
  T2->exists( t2 | t2.id = s2.id  and
                   t2 : t.tsub ) ) )
```

This looks up $T2$ elements $t2$ corresponding to the $s2$ in *ssub*, creating such elements if they do not already exist, and then adding them to *t.tsub*.

Related patterns This pattern can be used to improve Recursive Descent solutions (Section 5.9) by using implicit invocation instead of explicit invocation.

5.11 Introduce Rule Inheritance

Summary This pattern uses rule inheritance to factor out commonalities between several rules (e.g., those processing the subclasses of an abstract source entity type) or to support evolution of a transformation in response to evolution of the source language.

Application conditions This pattern is relevant if there is an abstract superclass A of source model classes B, C, and the processing of A's features are the same for each of its subclasses.

Solution Factor out common processing from the rules matching B or C elements of the source model, and define this processing as a rule or operation of A, which is inherited or invoked by the rules for B and C.

Benefits This pattern reduces duplicated specification text. It also facilitates the evolution of a transformation to deal with extended source metamodels (e.g., if a source entity S_j inherits a source entity S_i, and extends it by additional attributes, the transformation rule that maps S_j can inherit the original rule for S_i, and extend it with mappings for the new attributes).

Disadvantages This form of adaption is not very flexible because the inheritance mechanism introduces explicit named dependence of the subclass on the superclass.

Applications and examples This pattern is applicable to all categories of transformation.

Rule inheritance is used in ATL to extend existing rules with additional functionality; an example is given in [35]. In general the facility for rule inheritance would be a useful one for any model transformation language.

Related patterns The pattern is a specialized version of the classical Template Method pattern [18], and is analogous to the Pull-up Feature refactorings of [16]. An alternative way of factoring out common rule processing is to introduce a generic rule which can carry out the effect of different specific rules depending upon a parameter value, for example, the FunnyQT solution of [50].

6 Optimization Patterns

These patterns are concerned with improving the efficiency of a transformation implementation.

6.1 Unique Instantiation

Summary To avoid duplicate creation of objects in the target model, a check can be made that an object satisfying specified properties does not already exist, before such an object is created.

Application conditions Required when duplicated copies of objects in the target model are forbidden, either explicitly by some mechanism such as the $T_j \rightarrow exists1(t \mid Post)$ quantifier, the **create unique** keyword, or implicitly by the fact that T_j possesses an identifier (primary key) attribute.

Solution In the case of a specification **create unique** $t : T_j$ **satisfying** $t.id = x$ *and Post* where *id* is a primary key attribute, check if a T_j object with this *id* value already exists: $x \in T_j.id$ and if so, use the object (e.g., $T_j[x]$) to establish *Post* (if *Post* is already true, take no action). Otherwise (if T_j is concrete), create a new instance t of T_j and apply $t.id = x$ *and Post* to this instance. The same technique can be used for specified effects **create** $t : T_j$ **satisfying** $t.id = x$ *and Post*.

To implement a specification **create unique** $t : T_j$ **satisfying** *Post* for a concrete class T_j, where there is no equation $t.id = x$ in *Post*, test if $T_j \rightarrow exists(t \mid Post)$ is already true. If so, take no action, otherwise, create a new instance t of T_j and establish *Post* for this t.

In cases where uniqueness of a feature f which is not a primary key is required, over some collection *coll*, i.e.: $coll \rightarrow isUnique(f)$ should hold, the check for existence is $coll.f \rightarrow includes(x)$, and in this case the element with f value equal to x is retrieved by $coll \rightarrow select(f = x) \rightarrow any()$ or $coll \rightarrow any(f = x)$.

Benefits The pattern ensures the correct implementation of the rules. It can be used when we wish to share one subordinate object between several referring objects: the subordinate object is created only once, and is subsequently shared by the referrers.

This pattern assists in the decomposition of a transformation, enabling a target object to be created in one phase and referred to and modified by a subsequent phase.

Disadvantages The cost of checking for existing objects can be significant. Look up of objects by key values, using the Object Indexing pattern, can reduce this cost.

Applications and examples This pattern is applicable to all categories of transformation. Application examples are given in Sections 12, 13. The pattern is provided as an inbuilt implementation mechanism in UML-RSDS. The pattern is used in the 'check before enforce' matching semantics for QVT-R rules: target model elements are only created to satisfy a rule if no element already exists that satisfies the rule. The *unique lazy* rules of ATL are similar in their purpose: such rules use the internal ATL tracing mechanism to look up previously mapped target elements and return these, instead of creating a new target element. Similar mechanisms are the use of *equivalent/equivalents* in ETL, and *resolveIn/resolveoneIn* in QVT-O [34], [62]. The pattern would be a useful facility for any transformation language to provide.

Related patterns Object Indexing (Section 6.2) can be used to efficiently obtain an existing object with a given primary key value in the first variant of the pattern. The pattern facilitates the use of the Entity Merging pattern (Section 5.4).

Traces defined by Auxiliary Metamodel or inbuilt to a transformation language can be generally used to ensure unique instantiation, for example, before mapping an attribute *att* to a column, we can test

$$Attribute2Column \rightarrow exists(a2c \mid a2c.source = att)$$

to identify if the attribute has previously been mapped.

6.2 Object Indexing

Summary All objects of an entity are indexed by a primary key value, to permit efficient lookup of objects by their key.

Application conditions Required when frequent access is needed to objects or sets of objects based upon some unique identifier attribute (a primary key).

Lookup of objects by means of an expression of the form $E.allInstances() \rightarrow select(id = v) \rightarrow any()$, $E.allInstances() \rightarrow any(id = v)$ or $E.allInstances() \rightarrow select(v \rightarrow includes(id))$ can be inefficient, with a worst case time complexity proportional to the number of elements of E.

Solution Maintain an index map data structure *cmap* of the (partial single-valued) map type $IndType \nrightarrow C$, where C is the entity to be indexed, and *IndType* the type of its primary key. Access to a C object with key value v is then obtained by applying *cmap* to v: $cmap.get(v)$. When a new C object c is created, add $c.ind \mapsto c$ to *cmap*. When c is deleted, remove this pair from *cmap*. Figure 10 shows the structure of the pattern. The map *cmap* can be represented as a qualified association, and is an auxiliary metamodel element used to facilitate separation of the specification into loosely coupled rules. The C object with primary key value v is denoted by $C[v]$. The set of C objects with a key value in v is also denoted $C[v]$, if v is a set.

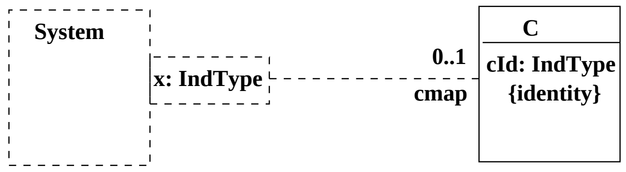

Fig. 10. Object Indexing pattern

Expressions $E.allInstances() \rightarrow select(id = v) \rightarrow any()$ or $E.allInstances() \rightarrow any(id = v)$ in the specification are then replaced by $E[v]$ for accessing single E objects by identity, and expressions $E.allInstances() \rightarrow select(v \rightarrow includes(id))$ for accessing sets of E objects are also replaced by $E[v]$, reducing the syntactic complexity of the specification. Multi-level indexing maps $IndType1 \nrightarrow (...(IndTypen \nrightarrow E)...)$ can be used to represent compound keys with n components.

Benefits The pattern substantially improves the efficiency of object lookup, making this a constant-time operation independent of the size of the domain C.

Syntactic complexity is also reduced. This pattern also assists in the separation of a transformation into phases.
Disadvantages The key value of an object should not be changed after its creation: any such change will require an update of *cmap*, including a check that the new key value is not already used in another object. Support for indexing needs to be provided by the transformation language. The use of indexing increases memory requirements.
Applications and examples The pattern is applicable to all categories of transformation. We have given examples of the use of this pattern in Sections 5.5 and 5.6.

The simple version of the UML to relational database transformation (Figure 2) can be expressed using this pattern, since *name* is a primary key for *RDBElement*:

```
for each $p : Package$
create $s : Schema$ satisfying $s.name = p.name$

for each $c : Entity$
create $t : Table$ satisfying $t.name = c.name$ and
          $t.schema = Schema[c.package.name]$

for each $a : Attribute$
create $cl : Column$ satisfying $cl.name = a.name$ and
          $cl.table = Table[a.owner.name]$
```

In the 2nd and 3rd rules the target container objects *t.schema* and *cl.table* are looked-up by primary key from the schemas and tables created by rules 1 and 2, respectively.

The concept of *key* attribute in QVT-R is another example of the use of this pattern. The pattern is a good candidate for inclusion as an inbuilt facility in any transformation language, due to its general applicability.
Related patterns The pattern can be regarded as an implementation-level application of the Auxiliary Metamodel pattern (Section 5.7). It can be used by Phased Construction and its specializations to look up the target elements corresponding to previously-processed source model elements. It can be used by Unique Instantiation (Section 6.1) to look up elements to avoid duplicating them. It is related to the Cache Management pattern of [19].

An alternative strategy to look up target model elements is to use an explicit transformation trace facility, as in Kermeta [32] and Viatra [56], or implicit traces as in ATL or ETL [34]. However, lookup using traces will be less efficient unless specialized support is provided by a transformation language, since *select*/*any* searches over the set of traces are necessary.

6.3 Omit Negative Application Conditions

Summary Redundant tests for negative application conditions (NACs) of a rule are omitted in order to optimize its implementation.
Application conditions If it can be statically deduced that the succedent of a rule is inconsistent with the antecedent, omit checks in the rule implementation for the truth of the succedent. Such checks can constitute a substantial part of the execution time of the transformation rule for complex succedents.
Solution Before applying a rule

```
for each $s : S_i$ satisfying $SCond$
do $Succ$
```

to source model elements, model transformation languages usually test to check if the succedent *Succ* is already true (if *SCond* holds): if this test succeeds, then the rule is not applied. However, if it is known that $Succ \Rightarrow \neg\,(SCond)$ then the test of *Succ* is unnecessary and can be omitted.
Benefits The executable code of the rule is simplified and its execution cost is reduced.
Applications and examples The pattern is applicable to all categories of transformation, but it is particularly relevant for update-in-place transformations such as refactorings. It is an inbuilt implementation technique in UML-RSDS.

An example of applying the pattern is the derivation of the root class of an *Entity* (Figure 7), where *rootClass* is a *..0..1 auxiliary association from *Entity* to *Entity*:

```
for each $e : Entity$ satisfying $e.parent \rightarrow size() = 0$
do $e.rootClass = Set\{e\}$
```

This rule sets the root class of each class *e* without superclasses to be *e* itself.

```
for each $e : Entity$
satisfying $e.parent \rightarrow size() > 0$ and
          $e.rootClass \rightarrow size() = 0$ and
          $e.parent.rootClass \rightarrow size() > 0$
do $e.rootClass = e.parent.rootClass$
```

This rule sets the root class of a class *e* with superclasses to be the root class(es) of all its direct superclasses, if *e*'s root has not already been set and if its parent does have a set root class. In the second rule, the succedent contradicts the antecedent, so the succedent test can be omitted from the rule implementation.

This example also illustrates a case of Sequential Composition (Section 5.5) where two rules write disjoint sets of objects, and the second reads data written by the first. The first rule here has a bounded iteration implementation, but the second needs a fixed-point implementation.
Related patterns The Replace Fixed-point by Bounded Iteration optimization (Section 6.4) is an alternative optimization of such rules.

6.4 Replace Fixed-point by Bounded Iteration

Summary Use a single bounded iteration over a source domain S_i, instead of a fixed-point iteration, in cases of rules based on S_i which also create or delete S_i elements, if each application strictly reduces the set of S_i elements that satisfy the application conditions of the rule.

Application conditions A rule

```
for each $s : S_i$ satisfying $SCond$
do $Succ$
```

can be implemented by a single iteration over S_i provided that each application of *Succ* reduces the set of elements that match the application conditions of the rule.

Solution For such a rule, if

$$\forall\, matches \cdot matches = \{s : S_i \mid SCond\} \;\wedge\; matches \neq \{\} \;\Rightarrow\; [stat(Succ)](\{s : S_i \mid SCond\} \subset matches)$$

can be proved, where $stat(Succ)$ implements *Succ* and $[act]P$ is the weakest-precondition operator on behaviors *act* and expressions P, then a linear iteration over the original set of S_i elements is sufficient to ensure that the rule is established. One test for this situation is that if $s' : S_i$ is created in *Succ*, e.g., by a formula $S_i{\rightarrow}exists(s' \mid P)$, then P should be inconsistent with $SCond[s'/s]$ *and* $SCond$.

In addition, the special case of deletion rules:

```
for each $s : S_i$ satisfying $SCond$
delete $s$
```

will also satisfy the application conditions of the pattern, provided the deletion of s cannot make $SCond[s'/s]$ true for an existing $s' : S_i$.

Benefits The execution time of the optimized rule implementation depends only upon the original number of elements of S_i, not upon the maximum number that could exist. There is no need for proof of termination of the implementation, e.g., using a variant function [45].

Applications and examples The pattern is applicable to all categories of transformation, but is particularly relevant for update-in-place transformations.

An example is the 'repotting geraniums' example of [61] (Figure 11):

```
for each $p : Pot;\ v$ satisfying
    $p.broken = true\ and$
    $v = p.plants{\rightarrow}select(flowering = true)\ and$
    $v \neq Set\{\}$
create $p1 : Pot$ satisfying $p1.plants = v$
```

For every broken pot p that contains some flowering plant, a new (unbroken) pot p' is created and the flowering plants of the broken pot are transferred to the new pot. Even though new pots are created by applications of the rule, the number of pots satisfying the application condition (the antecedent of the rule) is reduced by each application. $p1.broken = false$ follows from the succedent, so $p \neq p1$, and neither *SCond* or $SCond[p1/p]$ can be true after the application (otherwise $p1.plants \cap p.plants \neq Set\{\}$, a contradiction).

In UML-RSDS the pattern is an inbuilt implementation facility. Rules can be marked for optimization by this

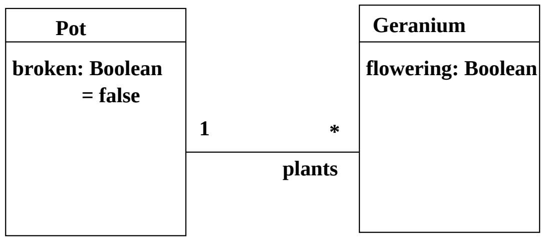

Fig. 11. Repotting geraniums metamodel

pattern by using pre-state versions of the entities and features involved on the LHS of the rule, e.g.:

```
for each $p : Pot@pre;\ v$ satisfying
    $p.broken@pre = true\ and$
    $v = p.plants@pre{\rightarrow}select(flowering = true)\ and$
    $v \neq Set\{\}$
create $p1 : Pot \cdot p1.plants = v$
```

for the above example, where *f@pre* is the value of the entity type or feature f at the start of execution of the transformation rule. This indicates that the rule does not write any data that it also reads (*Pot*, *plants* and *broken* are written, but only the pre-state versions of these features are read), so that a bounded iteration can be automatically selected for implementation of the rule.

In addition, if there are two mutually data-dependent rules $C1$, $C2$ where $C1$ reads entity type E, and $C2$ creates instances of E, we can assert that the instances created by $C2$ are irrelevant to $C1$ by using *E*@pre in $C1$, denoting the set of E objects created before $C1$ begins execution, thus potentially removing the need for fixed-point iteration of the rule group $\{C1, C2\}$.

6.5 Decompose Complex Navigations

Summary Simplify long navigation expressions in antecedent tests of a rule, in order to optimize its implementation.

Application conditions The pattern is relevant if there are one or more quantifier range expressions in the antecedent of a rule which are compositions of two or more collection-valued association ends, and only one element needs to be selected from the quantifier range.

A test $a : f1.f2$ which selects an element a of the composition of two or more collection-valued association ends is potentially inefficient to compute, because the composition may involve the construction of large collections of elements.

Solution Decompose the test into two or more simpler matching tests:

$$s : f1 \;and\; a : s.f2$$

where s is a new variable identifier, not occurring elsewhere in the rule.

Benefits If the size of $f1$ is typically M and that of $f2$ is N, then only $M + N$ elements are searched for an initial match for a in the optimized version, in contrast to $M*N$ elements in the unoptimized version.

Therefore, the optimization is beneficial particularly for rules for which there is a high likelihood of finding a matching a in any particular $s.f2$ set, and where only one element needs to be selected: this second condition is typically the case for fixed-point implementations of a rule. Both execution time and memory usage will potentially be reduced.

Applications and examples The pattern is applicable to all categories of transformation, but is particularly relevant to refactorings or other update-in-place transformations which modify some data that they also read: in such cases the implementation is a fixed-point iteration which selects an element from a set of candidates, applies the rule, and then selects another element, from a possibly changed set of candidates.

In the class diagram rationalization case study solution using UML-RSDS [33], two rules include an antecedent test `a : specialization.specific.ownedAttribute` to select some attribute a of a subclass of the current class. This can be optimized by writing `s : specialization.specific and a : s.ownedAttribute`.

Related patterns The *Usage of iterators* pattern in [11] considers the related problem of selecting any value in a collection that satisfies a condition, without computing the entire collection of such values. The Restrict Input Ranges pattern is another technique for reducing condition evaluation time, and may be used together with Decompose Complex Navigations.

6.6 Restrict Input Ranges

Summary Restrict the ranges of input variables in a rule, based upon necessary conditions which these must satisfy, in order to optimize the rule's implementation.

Application conditions The pattern is relevant if there are one or more quantifier range expressions in the antecedent of a rule which can be made more restrictive because of further matching conditions which the variables must satisfy for the rule to be applied. By restricting the quantifier ranges the rule can be made more efficient.

An indication that the pattern is relevant is if there are two or more input variables ranging over entity types:

```
**for each** $s1 : S_1;\ s2 : S_2$ **satisfying** $P$
...
```

Such a rule will have worst case time complexity at least $card(S_1) * card(S_2)$.

Solution A range expression $a : R$ which selects an element a of the entity type or collection R can be replaced by $a : e$, where e is a small subset of R, if the remainder P of the matching conditions implies that $a \in e$. In the above example, $s2 : S_2$ could be replaced by $s2 : s1.f$ where f is a feature or composition of features based on $s1$.

Benefits The number of elements that are searched for a match for a in the optimized version are less than in the original. Only elements that can possibly be a match are considered, irrelevant elements are not evaluated, reducing expression evaluation time.

Applications and examples The pattern is applicable to all categories of transformation.

In the movies database case study solution using UML-RSDS [51], the rule to find pairs of actors who have appeared in at least 3 movies together has the unoptimized form:

```
p : Person & q : Person &
p.name < q.name &
comm = p.movies /\ q.movies &
comm.size > 2    =>
    Couple->exists( c |
                    p : c.p1 & q : c.p2 &
                    c.commonMovies = comm )
```

This can be optimized by restricting the range of q from *Person* to *p.movies.persons* because the conditions $comm = p.movies \cap q.movies\ \&\ comm.size > 2$ imply that $q \in p.movies.persons$.

Related patterns The Decompose Complex Navigations pattern also aims to reduce the cost of searching input ranges, and both patterns can be applied together. The usual programming tactic of placing more restrictive conditions earlier in a series of conjuncts can also be used for the antecedents of rules, under the assumption that a 'shortcircuit' evaluation policy for conjunction is used in the implementation.

The pattern could evolve to Filter Before Processing (Section 9.5): a pre-filter could be introduced to discard from the input model all elements which could not satisfy any of the transformation rule conditions.

6.7 Remove Duplicated Expression Evaluations

Summary Avoid duplicated evaluation of expressions by using mechanisms to retain or cache their values.

Application conditions This pattern is relevant if the same expression evaluation occurs in multiple places within a transformation rule or specification, and these evaluations will produce the same value at these different locations. It also applies if the same form of complex expression reoccurs with different arguments.

Solution Remove duplicated evaluations of complex expressions from transformation specifications: duplicated expressions e within a single rule can be factored out by using a *let* $v = e$ construct, which evaluates the expression e once, the value can be subsequently referenced by the identifier v. Duplicates in different rules can be factored out by defining auxiliary data features and storing pre-computed expression values in these features. Alternatively, they can be factored (with or without caching) by defining query operations to compute them. For duplicated expression forms with varying arguments, formal parameters of the query operations are defined.

Benefits Modularity of the specification is increased, because of the higher factorization of the specification after the pattern is applied. Efficiency should usually be increased, if duplicated evaluations are removed.

Disadvantages Efficiency may not be increased in certain cases. If cached values are never used then the overhead of caching may increase memory use and execution time. Likewise, if functions are used to factor out repeated expressions, then there is an additional overhead of function calling.

Applications and examples The pattern is applicable to all categories of transformation. This optimization is performed in ATL by moving duplicated expression evaluations to helper operations, whose results are cached. Depending on the structure of the input model, this may reduce execution times [4]. The unique lazy rules of ATL provide another means of avoiding duplicated evaluations, as do ≪ *cached* ≫ query operations in UML-RSDS. Languages such as Kermeta and Viatra also have mechanisms for factoring out repeated evaluations.

An example of duplicated expressions within a rule occurs in the class diagram rationalization case study solution using UML-RSDS [33], in the rule:

```
a : specialization.specific.ownedAttribute and
v = specialization->select(
    specific.ownedAttribute->exists( b |
              b.name = a.name  and
              b.type = a.type ) )  and
v.size > 1   implies
   Entity->exists( e |
       e.name = name + a.name  and
       a : e.ownedAttribute  and
       e.specialization = v  and
       Generalization->exists( g |
           g : specialization  and
           g.specific = e  and
           v.specific.ownedAttribute->
                   select( name = a.name )->
                           isDeleted() ) )
```

operating on instances of *Entity*. Here, a let variable v is used to avoid re-computation of the complex expression

```
specialization->select(
    specific.ownedAttribute->exists( b |
                   b.name = a.name  and
                   b.type = a.type ) )
```

which is used in three places in the constraint.

An example of duplicated evaluations between rule applications is the rule

```
for each $c : Entity$; $a : c.attribute$
create $cl : Column$ satisfying
    $cl.name = a.name$ and
    $cl : Table[c.rootClass().name].column$
```

in the revised UML to relational database transformation (Section 5.4). Here, the computation of the *rootClass*() of each entity can involve duplicated evaluations (computing the root class of an entity involves computing the root class of all its superclasses), and it may be more efficient to precompute these and store them in an auxiliary association *rootClass* : *Entity* of *Entity* (as in Section 6.3), prior to the rules creating and updating tables.

The rule would then become:

```
for each $c : Entity$; $a : c.attribute$
create $cl : Column$ satisfying
    $cl.name = a.name$ and
    $cl : Table[c.rootClass.name].column$
```

An example of repeated expression forms is given in Section 12.

Related patterns This is a special case of a general modularization and optimization strategy for factoring out repeated sub-expressions from programs or specifications [38]. In [11] the related pattern *finding constant expressions* is described to factor out repeatedly evaluated expressions with constant values. Auxiliary Metamodel (Section 5.7) can be used to cache precomputed expression values in auxiliary metamodel elements. The concept of memoisation of function values [52] is another form of caching of result values to avoid redundant computations.

The same concept of factoring can be applied to common update functionality that is repeated in different rules. However, such factoring only improves the modularity and clarity of the specification, and does not improve efficiency. Implementation of update factoring using explicit rule invocation is described in [35].

6.8 Implicit Copy

Summary This is a pattern specific to migration transformations. The pattern avoids the explicit copying of identical source model structure to target model structure, by means of strategies for implicitly copying such structure.

Application conditions Useful when the source and target languages contain large parts of similar or identical structure. Rules which consist mainly of copying entity types and features are a sign that this pattern is needed. Without using the pattern, a large number of rules, at least one for each source language entity type, will be needed.

Solution Specify implicit copying of entity types and features from source to target by some mechanism such as the *conservative copy* strategy of Epsilon Flock [64]. If source entity type E is identified as corresponding to target entity type F, then all the same-named and same-typed features of E are copied implicitly by default to those of F. Only differently-named/typed features need to be mapped explicitly.

Benefits The pattern makes a migration transformation specification more concise by avoiding explicit copying of corresponding data features. The transformation may not need to change even if the source language changes, e.g., by a simple enlargement of the feature sets of entity types within the metamodel.

Disadvantages The implicit copy process needs to be made explicit for verification. Implicit copying may be semantically incorrect in some cases, e.g., there can be features with the same name and type in different entity types, but which have distinct meanings. Complex structural changes between the source and target (such as entity splitting) cannot be defined using this pattern.
Applications and examples This pattern is specifically relevant to migration transformations, although it could be used in other categories of transformation, except model-to-text or text-to-model.

The Flock solution to the migration problem of [64] is an example of this pattern. For example, the rule

```
migrate ActionState to OpaqueAction
```

specifies the implicit copy of all features of *ActionState* to matching features of *OpaqueAction*.

The UML-RSDS tools generate copying transformations, of the Structure Preservation form, from language morphisms of the source to the target languages (Appendix). The pattern is potentially useful as an inbuilt facility for general-purpose model transformation languages.
Related patterns This is one way of implementing the Structure Preservation pattern (Section 5.2), if the transformation language supports the implicit copy mechanism.

7 Model-to-Text Patterns

These patterns concern the generation of code or other text from models.

7.1 Model Visitor

Summary This is a version of the classic Visitor [18] design pattern, aimed at generating text from models in a systematic manner.
Application conditions This pattern is suitable for simple cases of text generation where a fixed model traversal strategy is adequate to generate text from the model.
Solution Define Visitor acceptance operations for each relevant model entity type, and a Visitor entity type (as an auxiliary metamodel entity type) to be passed around the model structure and to accumulate/generate text from it.
Benefits This pattern separates the model-to-text conversion process from the source model entity types.
Disadvantages The pattern requires a close correspondence between the model structure and the text structure. A fixed order of visiting the source language entity types is defined.
Applications and examples The pattern is primarily relevant for model-to-text transformations, although it could be used for other transformation categories.

The CodeWriters facility in Jamda uses this approach to generate code from UML models [28].

Related patterns This pattern is related to Recursive Descent (Section 5.9), since both define their processing based on the structure of source model elements. Model Visitor can involve a wider variety of traversal strategies than Recursive Descent. The pattern uses Auxiliary Metamodel (Section 5.7).

7.2 Text Templates

Summary Use literal text templates to generate text from models, where variables or operations within the template can access the source model to fill in specific values or to choose variations of the template.
Application conditions This pattern is suitable for a wide range of model-to-text transformations, where only text is needed as output, not a model representing the output document or program. If rules only output text, and contain substantial fragments of literal string text to form this output, this is a sign that the pattern is required.
Solution Use templates of target language constructs to produce the output text, accessing source elements to configure the output.
Benefits Arbitrary text can be generated, for program code or document content, without the need to construct a metamodel for the target language. For programming languages, such metamodels can be large and complex.
Disadvantages Verification of such transformations is difficult because the result is a linear sequence of text, without structure.
Applications and examples The pattern can be used for model-to-text transformations or higher-order transformations.

OpenArchitectureWare [55] (now part of the Eclipse Modeling Project, www.eclipse.org/modeling) uses this approach, providing templates that can be defined and invoked as subroutines, and executed in loops upon all source elements of a specified entity type. In [69] the idea of text templates is introduced to make higher-order transformations more concise: the text template describes in concrete syntax a transformation to be produced by the higher-order transformation. The EGL language is a specialized model-to-text language which uses templates (www.eclipse.org/epsilon/doc/egl).

An example text template transformation in UML-RSDS, outputting C# classes from UML models, could consist of two rules:

```
("using System; \n")->display()

("class " + c.name + "{ \n")->display()  and
ownedAttribute->forAll( at |
                     at.toCS()->display() )  and
"} \n"->display()
```

where the second rule iterates over *Entity* and *toCS*() is an operation of *Property* which returns the corresponding declaration text in C# for this property.

7.3 Replace Abstract by Concrete Syntax

Summary This pattern replaces predicates defining target model navigation construction and modification at the abstract syntax level by more concise specification using the concrete syntax of the target language. Concrete syntax of the source language can be used to specify transformation rules, if an unambiguous concrete syntax exists.

Application conditions Useful when source or target languages have complex metamodels (e.g., UML class diagrams, or programming languages), and which also have unambiguous concrete syntax.

Solution Specify rules using the concrete graphical or textual syntax of the source and target languages, instead of predicates or diagrams in terms of the abstract syntax of the languages.

Benefits Rule specifications in concrete syntax are usually considerably more concise and comprehensible than rules using abstract syntax, particularly if a graphical concrete syntax is used.

Disadvantages This pattern requires that unambiguous concrete syntax for the source and target languages exist, and that the model transformation language supports such specification. Graphical specification of logical operators, such as quantifiers, is difficult, and no agreed-upon standard representation of these exists.

Applications and examples This pattern is applicable to all transformation categories.

The CGT language described in [22] provides such rule definition facilities. Figure 12 shows an example concrete grammar rule for the activity diagram restructuring example of [22]. A state with a self-transition (LHS) and no other self-transition (NAC) is rewritten into a state with an internal loop action and no self-transitions (RHS).

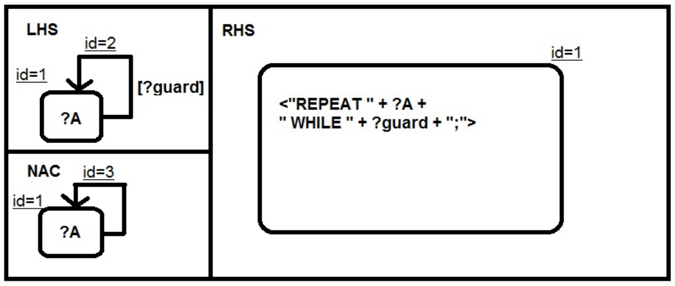

Fig. 12. Concrete syntax rule to remove self-loops

An equivalent abstract syntax rule could be:

```
for each $s : State$; $t : Transition$
satisfying $t.source = s$ and $t.target = s$ and
    $s.outgoing = Set\{t\}$ and $s.incoming = Set\{t\}$
do $s.code =$ “REPEAT” $+ s.code@pre +$
        “WHILE” $+ t.guard +$ “; ” and
    $t \rightarrow isDeleted()$
```

8 Expressiveness Patterns

These patterns define ways in which the limitations of certain kinds of transformation languages can be overcome.

8.1 Simulating Universal Quantification

Summary The pattern simulates an antecedent condition $X \rightarrow forAll(x \mid P)$ by a double negation $not(X \rightarrow exists(x \mid not(P)))$.

Application conditions This pattern is required when the transformation language does not support universal quantification in the conditions of rules. It especially applies to graph transformation languages such as GrGen.NET, where conditions are expressed as (existential) pattern matches in a graph.

Solution Express universal conditions in rule antecedents by their logical equivalents using negation and existential quantification.

Benefits The pattern enables universally quantified conditions to be expressed.

Disadvantages The indirect style of expression of universal quantifications, using double negation, hinders comprehension and verification.

Applications and examples This pattern is applicable to all transformation categories.

The example rule previously shown from the GrGen.NET solution to the class diagram rationalization problem [33] is also an example of this pattern:

```
1  rule rule1 {
2    c : Class;
3    :SuperOf(c,g1); :SuperOf(c,g2);
4    g1:Class -:ownedAttribute-> a1:Property
         -:type-> t : Type;
5    g2:Class -:ownedAttribute-> a2:Property;
6    :SameAttribute(a1,a2);
7    negative {
8      g3 : Class;
9      :SuperOf(c,g3);
10      g1;
11      negative {
12        g3 -:ownedAttribute-> a3:Property;
13        :SameAttribute(a1,a3);
14      }
15    }
16    modify
17      c -:ownedAttribute-> a4 : Property
           -:type-> t;
18      eval {
19        a4._name = a1._name;
20      }
21      exec(RemoveAttributeFromSubclasses(c,a4)
          ;> [createInverseEdges]);
22    }
23  }
```

The negative application condition (from lines 7 to 15 inclusive) simulates a universal quantification condition that all subclasses *g3* of *c* have an attribute equal to *a1*.

8.2 Simulating Explicit Rule Scheduling

Summary Use additional application conditions of rules to enforce relative orders of rule execution.

Application conditions Used when the order of application of different rules must be constrained, and the transformation language has no explicit mechanisms to achieve this (as, for example, in QVT-R).
Solution A rule $R2$ which should occur only after application of a different rule $R1$ is specified as

> **for each** $s : S_2$ **satisfying** *SCond*2 *and Post*1
> **do** *Succ*2

where *Post*1 is a condition which can only be established by $R1$.
Benefits The pattern enables control over the order of rule execution.
Disadvantages The simulation results in additional complexity for rules, and there is no explicit separation between the intrinsic conditions *SCond*2 of rules and the additional ordering conditions. The ordering conditions embedded in later rules may need to change if the effects of earlier rules are modified, resulting in poor maintainability.
Applications and examples This pattern is useful for purely declarative languages such as QVT-R which lack explicit rule ordering mechanisms (as provided in hybrid languages such as GrGen, Viatra and UML-RSDS). An example in QVT-R is a version of the UML to relational mapping:

```
top relation Package2Schema
{ checkonly domain uml p: Package { name = pn }
  enforce domain rdbms s: Schema { name = pn }
}

top relation Class2Table
{ checkonly domain uml c : Class
  { package = p : Package {}, name = cn }
  enforce domain rdbms t : Table
  { schema = s : Schema {}, name = cn }
  when
  { Package2Schema(p,s); }
  // p already mapped to s
}

top relation Attribute2Column
{ checkonly domain uml a : Attribute
  { owner = c : Class {}, name = an }
  enforce domain rdbms cl : Column
  { table = t : Table {}, name = an }
  when
  { Class2Table(c,t); }
  // c already mapped to t
}
```

In this case the additional ordering conditions are placed in the *when* clauses, and enforce that classes can only be mapped to tables after their respective packages and schemas have been mapped, likewise for attributes and columns. However, this does not necessarily ensure that *every* package has been mapped to a schema before any mapping of a class to a table can take place, and stronger *when* conditions would be needed to ensure this.
Related patterns Auxiliary metamodel elements may be introduced both to store the intermediate result of one rule for use by its successor(s), and to act as an enabler for these successor rules. For example, see the QVT-R specification of [33] and the GROOVE specification of [50].

9 Architectural Patterns

Architectural model transformation patterns are concerned with organizing systems of transformations in order to enhance the modularity, verifiability and efficiency of these systems. A particular concern is to reduce the size of models to be processed, by splitting input or output models and separately processing the separated model parts. We illustrate these patterns using UML activity diagrams to express systems of transformations.

9.1 Phased Model Construction

Summary Construct disjoint language parts of a target model successively from separate input models.
Application conditions The source and target models can each be split on the basis of some composition hierarchy of their entity types, such that the transformation can be carried out by mapping one part of the source model to one part of the target model, then by mapping the remainder of the source model to the remainder of the target model, and establishing any necessary links between the target model parts.
Solution Divide the input model $m : S$ into two disjoint or mainly disjoint models $m1 : S1$ and $m2 : S2$, where $S1$ and $S2$ are sublanguages of S and together make up S. The transformation is also split into successive parts which (i) map $m1$ to a subpart $n1$ of $n : T$ and then (ii) map $m2$ to a primarily disjoint subpart $n2$ of n (Figure 13).

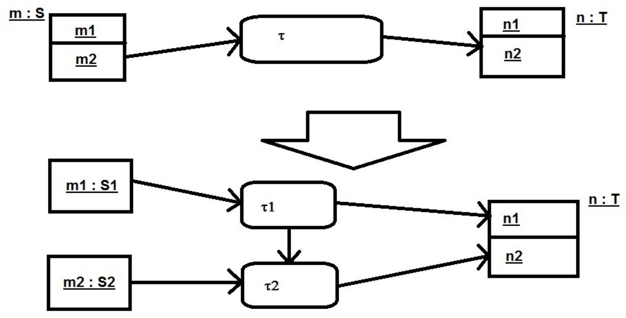

Fig. 13. Introducing Phased Model Construction

Benefits The input model size can be decreased by a factor based on the number of divisions of the model that are possible. The transformation is divided into smaller and potentially more cohesive transformations.
Disadvantages Model management is complicated by the division of a model into separate parts.
Applications and examples A small example is the UML to relational database example. We can split the source language into two parts: packages and classes, and classes and attributes. The corresponding divisions

of the target language are schemas and tables, and tables and columns.

The first transformation has two rules:

for each $p : Package$
create $s : Schema$ **satisfying** $s.name = p.name$

for each $c : Entity$
create $t : Table$ **satisfying** $t.name = c.name$ *and* $t.schema = Schema[c.package.name]$

The second has the single rule:

for each $a : Attribute$
create $cl : Column$ **satisfying** $cl.name = a.name$ *and* $cl.table = Table[a.owner.name]$

In this case the optimization of source language data storage is small: if there are N instances of each source language entity type in a source model, then the divided models each have $\frac{2}{3}$ of the number of elements of the original source model. However, if there are 20 source language entity types, each with N objects, a division with 11 entity types in each sublanguage gives a nearly 50% size reduction.

Related patterns The pattern is a further progression of Phased Construction (Section 5.1). The local nature of navigations used in Phased Construction facilitates the division of processing across source and target model parts. Sequential Composition can also be used to split a transformation into two or more successive subtransformations, which each only need a subset of the input data, thereby reducing memory requirements.

9.2 Target Model Splitting

Summary Construct separate target models successively from an input model, embedding implicit references to relate the target models.

Application conditions This pattern can be applied to transformations which create primarily separate and self-contained parts of a single target model.

Solution Divide the target language T into two disjoint sublanguages $T1$ and $T2$. The transformation is also split into successive parts which first map an input model $m : S$ to a model $n1 : T1$ and then map m to a model $n2 : T2$ (Figure 14).

Benefits The target model size can be decreased by a factor based on the number of divisions of the model that are possible. The transformation is divided into smaller and potentially more cohesive transformations.

Disadvantages Model management is complicated by the division of a model into separate parts. Relations between elements in the separate target models must be encoded by using foreign/primary key values. This may lead to greater effort being required for user comprehension and for data correlation.

Applications and examples An example could be the generation of SQL schemas and Java code of an application from the same UML class diagram model.

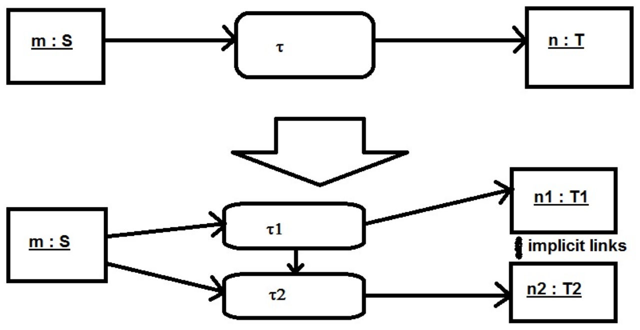

Fig. 14. Introducing Target Model Splitting

The two target models can be stored separately, they are implicitly connected by the common underlying language of entity names (class names and table names in the two target models) and feature names (instance variable names and column names).

Related patterns The pattern is a further progression of Entity Splitting (Section 5.3).

9.3 Model Merging

Summary Construct a target model from separate source models, using successive transformations: the first instantiates elements of the target model using the first source model, the second adds information to these elements from the second source model.

Application conditions This can be applied to transformations which process two or more separate and self-contained parts of a single source model, to populate a single target model.

Solution Divide the source language S into two sublanguages $S1$ and $S2$. The transformation is also split into successive parts which first map an input model $m1 : S1$ to instantiate elements of a model $n : T$ and then map $m2 : S2$ to enhance this model. (Figure 15).

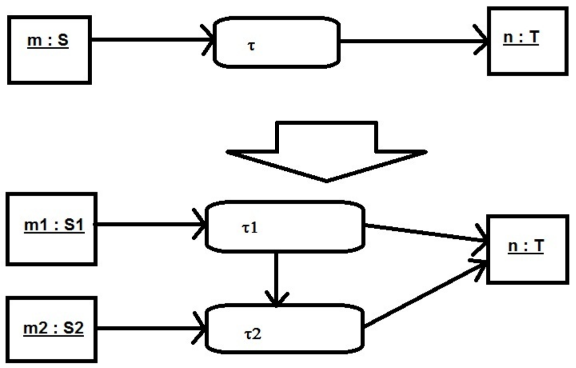

Fig. 15. Introducing Model Merging

Benefits The source model size can be decreased by a factor based on the number of divisions of the model that are possible. The transformation is divided into smaller and potentially more modular transformations.

Disadvantages Model management is complicated by the division of a model into separate parts. Care must be taken that the second transformation does not semantically interfere with the first, e.g., by overwriting data set by the first.

Applications and examples An example could be the merging of separate patient databases, one containing GP information on sets of patients, another containing hospital records of patients.

Related patterns The pattern is a further progression of Entity Merging (Section 5.4). Pre-filtering as in Filter Before Processing (Section 9.5) can be used if only part of the information in one or other input model is relevant to the transformation.

9.4 Auxiliary Models

Summary Decompose a transformation sequentially by introducing a new auxiliary or intermediate model to simplify transformation processing.

Application conditions This can be applied to simplify transformations which operate on complex input models by adding a preliminary transformation which produces an intermediate representation upon which the main transformation can more effectively operate (e.g., by deriving a semantically-oriented representation from a syntax-oriented representation, or by re-expressing some complex constructs in terms of simpler ones). It may also be introduced to facilitate evolution of a transformation to adapt an existing transformation to an enlarged source metamodel without changing the existing transformation.

It can be used to sequentially decompose a transformation into subtransformations which can be specified and implemented using different transformation languages and technologies, appropriate for the subtransformations.

Solution Introduce an intermediate language $S1$ and divide τ into a preliminary transformation $\tau 1$ which produces an intermediate auxiliary model $m1 : S1$, and a main transformation $\tau 2$ operating on $m1 : S1$ to achieve the effect of the original transformation (Figure 16).

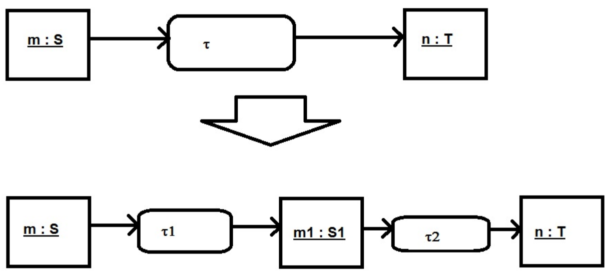

Fig. 16. Introducing Auxiliary Models

Benefits The transformation is made more modular and its processing is potentially simplified. The auxiliary model may be used by additional transformations.

Disadvantages Model management is complicated by the additional model and metamodel.

Applications and examples The concept of an intermediate language or *interlingua* is an example of this pattern, whereby M*N pairwise mappings between M sources and N targets are replaced by M mappings from the sources to a common intermediate representation, and N mappings from this representation to the targets.

An example is given in [23] of an application of this pattern, to factor a complex UML-to-PROMELA translation into two sequential steps, using a simplified intermediate representation of the UML source model semantics to facilitate the translation.

A UML to Java translation could be simplified by using a preliminary stage to reduce complex constructs such as association classes or multiple inheritance to constructs in a simpler UML sublanguage which can be more directly mapped to Java. In the UML-RSDS toolset, the translation from UML to B uses an intermediate representation in a metamodel for B, as an intermediate step in generating B textual models. The translation from UML to Java uses an intermediate activity model representation, this activity model is also used as the source for mappings to C# and C++ [47].

Related patterns The pattern is a further progression of Auxiliary Metamodel (Section 5.7): in Auxiliary Metamodel the auxiliary metamodel language is used internally within a transformation, whilst in this pattern it is defined externally as a separate language which is used as an input and output for subtransformations of the transformation.

The Model-Driven Architecture (MDA) concept of successive PIM to PSM and PSM to code transformations is related to this pattern.

9.5 Filter Before Processing

Summary When combining large models to derive information from selected elements of the models, prefilter one or both to reduce their size by selecting only the elements that are relevant for the combination.

Application conditions This can be applied to transformations which combine separate source models to derive selected information in a target model. It can also be applied to transformations operating on a single model, to filter out elements of the model that are irrelevant to the transformation. The pattern is appropriate if significant size reductions in input models can be achieved by discarding irrelevant data.

Solution Introduce a filter transformation which produces a smaller input model by using implicit information about what data is relevant to the combination transformation: other data is filtered out (Figure 17).

For transformations τ with a single input model, m, elements of m which cannot match or be navigated by a

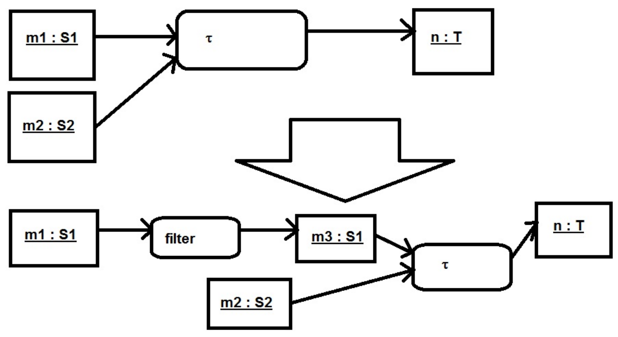

Fig. 17. Introducing Filter Before Processing

rule of τ can be pre-filtered to produce a smaller input model $m1$.

Benefits The source model size can be decreased substantially, as can the processing cost of the main transformation.

Disadvantages A new model and transformation need to be introduced; the filter transformation depends upon the second input model or upon the main transformation. The filter may involve fixed-point iteration because of cascaded deletion effects.

Applications and examples An example could be searching for N-sized groups of actors/actresses who perform together in three or more movies [25]: the input model $m1$ can be filtered to remove films with less than N cast members, and to remove actors who perform in less than 3 films.

Related patterns Construction and Cleanup is a dual to this pattern: a cleanup transformation post-filters an output model after the main transformation processing.

10 Combinations of Patterns

The patterns we have described in this paper are interrelated and can often be used in combination. Some patterns such as Filter Before Processing apply at the level of systems of transformations, others such as Phased Construction apply at the level of an entire transformation, whilst other patterns such as Restrict Input Ranges apply at the level of individual rules. Patterns at different levels can be used together to structure and optimize a transformation system.

Figure 18 shows the specialization and usage relationships between patterns, denoted by UML classes.

The following are the specialization relationships:

- Phased Construction is specialized by: Structure Preservation, Entity Splitting/Structure Elaboration, Entity Merging/Structure Abstraction, Map Objects Before Links.

The following patterns are often used together:

- Phased Construction (and its specializations) can use Object Indexing to look up elements created by an earlier rule to use them in a later rule. Multiple lookups of the same element in one rule can be factored out by introducing a let variable according to Remove Duplicated Expression Evaluations. Unique Instantiation can also use Object Indexing to look up elements to avoid recreating them.
- Map Objects Before Links uses Entity Merging to separate creation and update of target model objects.
- Entity Merging can use Unique Instantiation to avoid creating duplicate target objects.
- Remove Duplicated Expression Evaluations can use Auxiliary Metamodel to cache expression values referenced in several rules.
- Simulate Explicit Rule Scheduling can use Auxiliary Metamodel to store intermediate results and to control the enabling of rules.
- Structure Preservation can be implemented by Implicit Copy.
- Model Visitor uses Auxiliary Metamodel.

The following are the evolution relationships:

- Phased Construction can evolve to Sequential Composition: successive groups of rules which write to disjoint sets of entities and features can be factored out as separate transformations. It can evolve to Phased Model Construction if there is a need to reduce source model sizes.
- Entity Splitting can evolve to Parallel Composition/Sequential Composition: the internal decomposition of Entity Splitting could be extended to an external sequential or parallel decomposition if the additional data dependency restrictions of the latter patterns hold. It can evolve to Target Model Splitting if there is a need to reduce target model sizes.
- Entity Merging can evolve to Model Merging, if there is a need for separate input models.
- Auxiliary Metamodel can evolve to Auxiliary Models, particularly if the auxiliary data is useful as an input model to other transformations.
- Restrict Input Ranges could evolve to Filter Before Processing if a pre-filtered model is useful as an input to several transformations.
- Omit Negative Application Conditions can be refined to Replace Fixed-point by Bounded Iteration: by more detailed analysis the stronger optimization could be introduced.

The network of interconnections between the patterns described here, and the variation in scale of application of the patterns, indicates that the patterns form a *pattern language* for refinement, migration and refactoring transformations [3].

11 Selection of Patterns

In this section we give guidance on how to select patterns for transformations. Firstly we give general guidelines based on the category of a transformation, then we give specialized guidance based on quality measures of transformations and on metamodel structures.

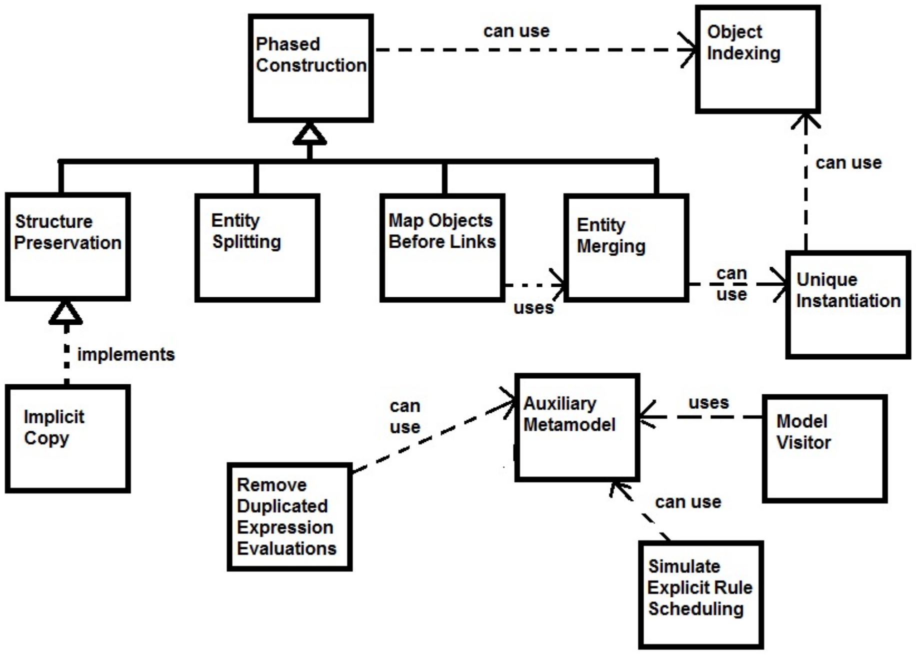

Fig. 18. Relationships between transformation design patterns

11.1 General guidance on pattern selection

When deciding how to construct the specification of a new transformation problem, a transformation developer should first clarify which category of transformation the problem requires, and then use patterns appropriate to that category. For input-preservation transformations, the Phased Construction pattern and its specializations are the preferred choices, with Recursive Descent only being used if the nature of the problem requires it. Parallel Composition and Auxiliary Metamodel can be used to sub-divide the processing into more modular parts, and other patterns, such as Unique Instantiation, can also be employed to solve specific problems. The optimization patterns Restrict Input Ranges and Remove Duplicated Expression Evaluations are relevant for input-preserving transformations.

For update-in-place transformations, Parallel Composition/Sequential Composition should be used to reduce to a minimum the scope of fixed-point implementation by grouping together those rules which have mutual data dependencies, and separating such groups from other rules. For example, if rules $r1$, $r2$ and $r3$ have $r1 < r2$, $r2 < r3$, $r3 < r1$, but rules $r4$, $r5$ have $r4 < r5$, $r1 < r4$, $r3 < r5$, with no other dependencies present, then two groups can be defined: $\{r1, r2, r3\}$ with a fixed-point implementation, sequentially followed by $\{r4, r5\}$, which does not require fixed-point implementation. An example of such a decomposition is given in Section 13. Optimization patterns such as Replace Fixed-point by Bounded Iteration can be used for update-in-place transformations to further remove non-essential cases of fixed-point iteration.

For migration transformations, Structure Preservation and Implicit Copy are particularly relevant. For model-to-text transformations, Model Visitor, Text Templates and Replace Abstract by Concrete Syntax are particularly relevant. Table 1 classifies the patterns for individual transformations, and identifies what categories of transformations they are suitable for.

The architectural patterns are relevant to all categories of transformation.

11.2 Transformation measures

In describing the patterns, we have given heuristics for their selection under 'Application conditions'. By formally defining metrics of model transformation specifications, it is possible to give precise guidance to transformation developers about which patterns are appropriate to apply, and to measure the benefits obtained by applying a pattern.

Suitable measurements (of complexity, modularization, factorization, etc) of a specification can identify that quality problems or 'bad smells' in the sense of [16] are present in the specification (e.g., that one rule in a model transformation has excessive complexity, or has a bad characteristic such as alternations of quantifiers in its postcondition) and can identify a number of patterns that could be introduced to restructure the specification and to improve its quality, by reducing the complexity measures. The pattern which maximally reduces the chosen measure could be selected to use. However, alternative patterns could also be considered, since they may enable further improvements by subsequent pattern applications. Likewise for designs.

Specifically, we can consider the following measures of model transformations τ:

Pattern	*Classification*	*Transformation categories*
Phased Construction	Modularization	Refinement, abstraction, migration
Structure Preservation	Modularization	As above
Entity Splitting	Modularization	As above
Map Objects Before Links	Modularization	As above
Entity Merging	Modularization	Also model merging
Parallel Composition	Modularization	All categories
Auxiliary Metamodel	Modularization	All categories
Construction and Cleanup	Modularization	All categories
Recursive Descent	Modularization	All categories
Replace Explicit by Implicit	Modularization	All categories
Introduce Rule Inheritance	Modularization	All categories
Unique Instantiation	Optimization	All categories
Object Indexing	Optimization	All categories
Omit Negative Application Conds.	Optimization	Refactoring, update-in-place
Replace Fixed-point by Bounded Iteration	Optimization	As above
Decompose Complex Navigations	Optimization	As above
Remove Duplicated Expression Evaluations	Optimization	All categories
Restrict Input Ranges	Optimization	All categories
Implicit Copy	Optimization	Migration
Model Visitor	Model-to-text	Model-to-text
Text Templates	Model-to-text	Model-to-text, higher-order
Replace Abstract by Concrete Syn.	Model-to-text	All categories
Simulating Universal Quantification	Expressiveness	All categories
Simulating Explicit Rule Scheduling	Expressiveness	All categories

TABLE 1
Model transformation pattern classification

1) Syntactic complexity 1: count of entity type references and feature references in a rule.
2) Syntactic complexity 2: count of operator occurrences in a rule.
3) Alternation of quantifiers: count of nestings of quantifiers of the form $\forall\exists\forall$ in rule postcondition/effect clauses.
4) Multiple creation: count of number of distinct element creation actions within a rule.
5) Duplicated expression occurrences: count of number of duplicated expressions within a rule (or in a complete specification) above a specified minimum syntactic complexity.
6) Maximally complex read subexpression: the syntactic complexity (measure 1 + measure 2) of the most complex subexpression read in the rules.
7) For each source entity type in a migration transformation, the proportion of source features f which are simply copied to a target language feature: $t.g = s.f$.
8) For each target entity type, a count of the number of rules that create/modify its elements.
9) The number of self-associations or cyclic structures of associations in the source language which are mapped to the target language.
10) The number of expressions in the transformation rules with the form $E.allInstances()\rightarrow select(id = v)$, $E.allInstances()\rightarrow any(id = v)$, or $E.allInstances()\rightarrow select(v\rightarrow includes(id))$ where id is a primary key of entity type E.
11) The number of input mapping ends of rules whose ranges (specified in the antecedent of rules) consist of navigations through two or more collection-valued features.
12) The maximum number N of slices of τ based on disjoint subsets of its write frame, which can be organized as separate transformations into a loop-free activity using only sequential and parallel composition. This is a measure of the (lack of) cohesion of τ. A cohesion measure $cohe(\tau)$ is the reciprocal of this measure.
13) Multiple entity input ranges: a count of the number of rules with two or more input variables whose specified range is an entity type.
14) Size, measured as the number of lines of specification text.

The sum of 1 and 2 over all rules of a transformation τ is taken as the overall syntactic complexity $comp(\tau)$ of the transformation. The average complexity of rules in a transformation, $avcomp(\tau)$, is also a useful measure.

An alternation of quantifiers value greater than 0 suggests applying the Phased Construction pattern to the rule. A multiple creation value greater than 1 also indicates this pattern, or the Entity Splitting pattern (depending on how the targets are derived from the source). A duplicated expression occurrence count $>$ 1 suggests the Remove Duplicated Expression Evaluations pattern: by using let expressions if the duplicated expressions occur only in one rule application, or by precomputation and the Auxiliary Metamodel pattern if they occur in multiple rules/rule applications. Multiple copies of update expressions suggests factoring using called auxiliary operations. A high value of maximum complexity for a read subexpression also suggests the Auxiliary Metamodel pattern, even if there is only one occurrence of the subexpression. In each case, applying a rule modularization pattern should reduce the complexity metrics 1 and 2 for the selected rule, and not increase the metrics of any other existing rule.

If there is a high proportion of feature copying from source to target, for migration transformations, then the Structure Preservation and Implicit Copy patterns are relevant. If a target entity type is updated by more than 1 rule, this indicates use of the Entity Merging pattern for this entity type. If there are cases of self-associations or cyclic association structures in the source language being mapped to the target, then Map Objects Before Links or Recursive Descent is relevant for this part of the transformation.

If there are any occurrences of $E.allInstances()\rightarrow select(id = v)$, $E.allInstances()\rightarrow any(id = v)$ or $E.allInstances()\rightarrow select(v\rightarrow includes(id))$ expressions in rules, this suggests use of Object Indexing for E, which will reduce the syntactic complexity of the rules containing the expressions.

If there are any quantifier range navigations through multiple collection-valued features in the antecedent of a rule, this indicates applying the Decompose Complex Navigations pattern.

If the cohesion measure is < 1, this suggests using Parallel and/or Sequential Composition, or an architectural pattern.

If there are multiple entity input ranges in a rule, the Restrict Input Ranges pattern may be relevant for the rule.

11.3 Target element creation and deletion orderings

Table 2 shows how creation and deletion of linked target model elements should be carried out, using the Phased Construction, Entity Splitting or Construction and Cleanup patterns. Mandatory association ends m are those with multiplicities such as 1 or $1..*$, i.e., with a lower bound greater than 0, whilst optional ends o are those with lower bound 0. In the case that both ends are mandatory, joint construction or joint deletion of elements linked by the association should be specified in a single rule, otherwise separate rules can be used. Whenever an instance of an entity type with a mandatory (opposite) association end is created, it should be linked to an instance by this association in the same rule, otherwise creation and linking could be performed in separate rules.

Association	*Construction*	*Destruction*
$A\ m$—$o\ B$	Create A instances before B instances. Link $b : B$ to A object(s) when b is created.	Delete B instances before A instances. Unlink $b : B$ from A object(s) when b is deleted.
$A\ m$—$m\ B$	Create and link A instances with B instances.	Delete and unlink B instances with A instances.
$A\ o$—$o\ B$	Create A instances before or after B instances, link after both created	Delete B instances before or after A instances, unlink before/ with first deletion

TABLE 2
Creation and deletion orderings

By using the metrics and rules discussed in this section, it should be possible to offer guidance to transformation developers on the selection and application of transformation patterns, and therefore to semi-automate the process of transformation improvement.

12 Illustrative Example 1: Lambda Calculus Re-expression Transformation

This transformation was set as the 'live challenge' problem at the Transformation Tool Contest (TTC) 2010 [21]. The problem is to map one representation of lambda calculus terms (Figure 19) to an alternative representation (Figure 20). This is an example of a 'syntax to semantics' mapping transformation, where a representation that corresponds closely to the concrete syntax of a notation is enriched by analysis to give a more semantically-oriented representation.

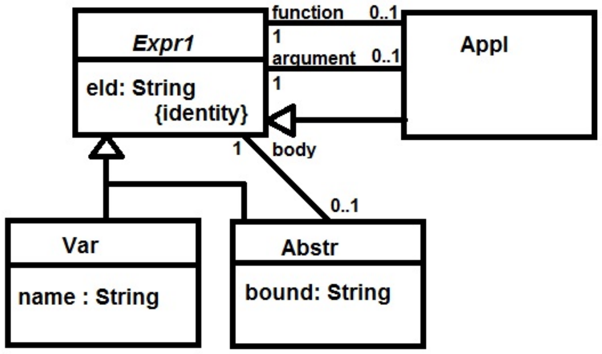

Fig. 19. First lambda-calculus representation

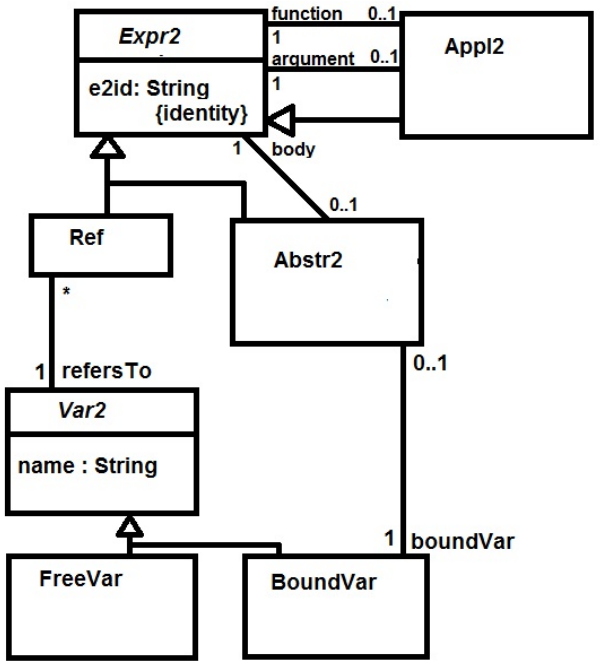

Fig. 20. Second lambda-calculus representation

The key differences between the two lambda-calculus representations are:

1) The first representation simply stores variable occurrences as strings, and does not correlate different occurrences of the same variable. The second

representation links, using *refersTo*, bound occurrences of a variable to the abstraction which binds it.

2) The second representation also distinguishes free variables from bound variables, and defines classes *FreeVar* and *BoundVar* to represent these. The class *Ref* holds variable references, which point to the relevant free or bound variable.

The recursive structure of the metamodels suggests use of the Map Objects Before Links pattern. However, further analysis of the problem identifies that the processing of subexpressions of an expression is dependent upon their surrounding context. For example, to process the subexpression $x\ y$ in $\lambda x.\ \lambda z.z\ (x\ y)$ we need to know that a lambda abstraction on x encloses the subexpression: the occurrence of x is then linked to the binding abstraction, whilst the occurrence of y is linked to the free variable with this name.

Therefore, the Recursive Descent pattern is appropriate. In addition, the need to retain the set of bound variables in scope for each subexpression when processing an expression, indicates also that the Auxiliary Metamodel pattern is useful. Unique Instantiation is needed, because there should be at most one free variable with a given name within an expression.

The transformation is therefore expressed procedurally, using operations with postconditions. For *Var*, *Abstr* and *Appl* an operation *mapToExpr*2(*bvars* : *Set*(*BoundVar*)) : *Expr*2 is defined, which maps the *Expr*1 expression to its equivalent *Expr*2 expression. These operations are themselves Auxiliary Metamodel elements. We will use the ASCII notation of UML-RSDS [47] to describe the transformation operations. For *Var*, the operation returns a *Ref* instance, which either points to a *BoundVar*, if the variable is an occurrence of a variable bound in the current scope (i.e., in *bvars*), or to an occurrence of a *FreeVar*. In either case, we need to use the Unique Instantiation pattern, because multiple instances with the same name should not exist in *bvars*, nor multiple instances of *FreeVar* with the same name. Since *name* is not a primary key of *Var*2, we must use an explicit lookup using *select* to obtain a pre-existing *Var*2 element with a given name:

```
bvars->select( b |
                b.name = name )->any()
```

for bound variables, and

```
FreeVar->select( f |
                   f.name = name )->any()
```

for free variables. To record the derivation links between the old and new forms of expression, new identity attributes *eId* and *e*2*Id* are introduced into *Expr*1, *Expr*2 respectively. This is an example of using Auxiliary Metamodel to define traces.

The definition of *mapToExpr*2 on *Var* is therefore:

```
mapToExpr2(bvars : Set(BoundVar)) : Expr2
pre: true
post:
 (name : bvars.name  =>
   Ref->exists( r |
    r.refersTo = bvars->select( b |
                 b.name = name )->any() &
    r.e2Id = eId & result = r ) ) &
 (name /: bvars.name =>
   ((name : FreeVar.name =>
       Ref->exists( r |
          r.refersTo = FreeVar->select( f |
                f.name = name )->any() &
          r.e2id = eId & result = r ) ) &
    (name /: FreeVar.name =>
       Ref->exists( r |
          FreeVar->exists( fv |
                   fv.name = name &
                   r.refersTo = fv ) &
            r.e2Id = eId & result = r ) ) ) )
```

The postcondition has a high syntactic complexity (70). There is a repeated expression form $s \rightarrow select(x \mid x.name = name) \rightarrow any()$ of size 5 in the postcondition (for the Unique Instantiation pattern), which can be factored out using the Remove Duplicated Expression Evaluations pattern: for this pattern we introduce an auxiliary query function

```
getByName(s : Set(Var2), n : String): Var2
pre: n : s.name
post:
  result = s->select( name = n )->any()
```

defined on *Var*2. The function is called in place of the repeated *select* expression. Therefore, the *mapToExpr*2 definition on *Var* can be simplified to:

```
mapToExpr2(bvars : Set(BoundVar)) : Expr2
pre: true
post:
 (name : bvars.name  =>
   Ref->exists( r |
     r.refersTo = Var2.getByName(bvars,name) &
     r.e2Id = eId & result = r ) ) &
 (name /: bvars.name =>
    ((name : FreeVar.name =>
        Ref->exists( r |
          r.refersTo =
               Var2.getByName(FreeVar,name) &
          r.e2Id = eId & result = r ) ) &
     (name /: FreeVar.name =>
        Ref->exists( r |
           FreeVar->exists( fv |
                    fv.name = name &
                    r.refersTo = fv ) &
           r.e2Id = eId & result = r ) ) ) )
```

This pattern application has reduced the syntactic complexity of the postcondition to 64.

On *Abstr* the *mapToExpr*2 operation is:

```
mapToExpr2(bvars : Set(BoundVar)) : Expr2
pre: true
post:
  Abstr2->exists( a2 |
    BoundVar->exists( bv |
      bv.name = bound & a2.boundVar = bv &
      a2.body =
         body.mapToExpr2(
            bvars->reject(name = bound)->
```

```
                    including(bv)) ) &
      a2.e2Id = eId & result = a2 )
```

The body of the abstraction is analyzed using *bvars* with any existing bound variable with name *bound* removed, and with *bv* added.

For *Appl* the operation is simply applied recursively to the subexpressions:

```
mapToExpr2(bvars : Set(BoundVar)) : Expr2
pre: true
post:
  Appl2->exists( a2 |
    a2.function = function.mapToExpr2(bvars) &
    a2.argument = argument.mapToExpr2(bvars) &
    a2.e2Id = eId & result = a2 )
```

This case study shows that transformation patterns are applicable even to relatively simple transformations, and can help to rationalize the transformation specification.

13 Illustrative Example 2: Petri-Net to Statecharts Mapping

This transformation was one of the cases in the TTC 2013 Transformation Tool Contest [70]. It maps Petri-Net models into hierarchically-structured statechart models, grouping Petri-Net places together into composite states on the basis of equivalent behavior. It involves complex update-in-place processing where the source model is progressively reduced in parallel with the construction of the target model.

Figures 21 and 22 show the transformations (represented as use cases) and the source and target language metamodels which we used in our UML-RSDS solution [49].

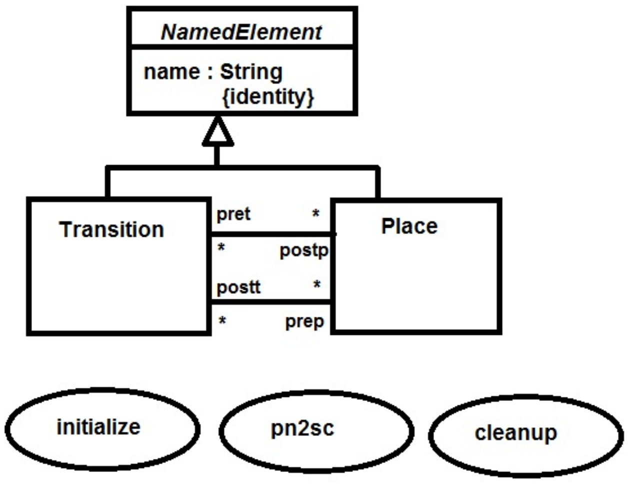

Fig. 21. Petri-Net metamodel

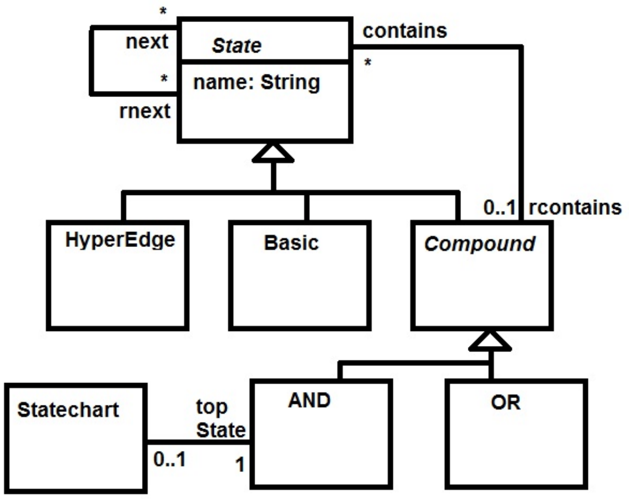

Fig. 22. Statechart metamodel

In solving this case, we used the following patterns:

1) Sequential Composition: the transformation is divided up into a sequence *initialize*; *pn*2*sc*; *cleanup* where *initialize* and *cleanup* are input-preserving, *pn*2*sc* updates both source and target models, whilst *initialize* and *cleanup* only update the target. *initialize* does not need a fixed-point implementation whilst the other sub-transformations do.
2) Map Objects Before Links: in the initialization phase, Petri-Net places are mapped to basic states and OR-states, and Petri-Net transitions are mapped to hyper-edges. The next/rnext links are then established between the states and hyper-edges based on the postt/prep and postp/pret links in the Petri-Net, e.g., rule (*I*3) has this form:
```
t : postt   =>
  HyperEdge[t.name] : Basic[name].next
```
applied to *Place* ("if *t* is a post-transition of self, then the hyperedge corresponding to *t* is in the next states of the basic state corresponding to self").
A reverse transformation can be directly derived from these initialization rules.
3) Object Indexing: in all stages of the transformation we made use of the uniqueness of *name* for the individual concrete State subclasses in order to lookup the state objects by name, e.g.: *OR*[*q*.*name*] in the OR-reduction rule of *pn*2*sc* to find the OR state corresponding to Place *q*.
4) Omit Negative Application Conditions: the three main reduction/construction rules of *pn*2*sc* all rewrite the Petri-Net and Statechart in such a way that the rule effect contradicts the rule assumption conditions, in particular, each rule deletes at least one of the Petri-Net elements that they matched on, so the Omit Negative Application Conditions pattern can be used to improve efficiency.
5) Construction and Cleanup: after the reduction/construction process has finished, there may exist redundant states which were created during the process but are now not needed. The *cleanup* transformation removes such states, e.g., to delete OR states with empty contents:

```
contains.size = 0 => self->isDeleted()
```
on *OR*.

The transformation is a substantial and complex example, whose specification has been made more systematic and comprehensible by the use of transformation patterns.

14 Evaluation

In our experience the patterns we have described here arise in many cases of refinement, migration, abstraction and refactoring transformations. From examination of published case studies many examples where these patterns have been used or could be used to improve a transformation specification or implementation have been identified. The patterns are closely inter-related and satisfy the criteria for a pattern language [3]. We do not consider patterns for model merging, model comparison or higher-order transformations, or for transformation aspects such as bidirectionality and change-propagation, and further work is needed to identify and document patterns for such transformation categories and aspects.

The following patterns are widely applicable and would be good candidates for inclusion as intrinsic features of model transformation languages: Auxiliary Metamodel, Introduce Rule Inheritance, Unique Instantiation, Object Indexing, Implicit Copy.

Table 3 summarizes improvements to measures of complexity, execution time or modularity which the application of the patterns of this paper can provide.

An example has been given in Section 5.1 of the reduction of average rule complexity by application of Phased Construction. In general, the pattern will achieve this reduction by replacing a constraint $C1$ of complexity $n1$ by two smaller constraints $C2$, $C3$ with complexities $n2 < n1$ and $n3 < n1$. This is also the case for the Entity Splitting and Map Objects Before Links patterns. Maximum rule complexity may also be reduced, although the total complexity and size of a transformation may increase.

Parallel Composition increases the cohesion measure $\frac{1}{N}$ of a transformation τ, where N is the number of activity nodes derived from disjoint slices of τ: dividing τ into $\tau 1$, $\tau 2$ on the basis of write frames leads to cohesion measures $\frac{1}{N1}$, $\frac{1}{N2}$ for these subtransformations, where $N = N1 + N2$. This is also the case for Entity Splitting.

Implicit Copy is shown to reduce size in the case study of [64], where the Flock specification is the smallest of the solutions.

Object Indexing replaces a search $E{\rightarrow}select(id = v){\rightarrow}any()$ or $E{\rightarrow}any(id = v)$ of worst-case time complexity $card(E)$ by a constant-time map application $E[v]$. In one experiment using a transformation from UML to J2EE with a model consisting of 1000 *UseCase* instances and 10,000 *Actor* instances, with 10 actors for each use case, the transformation using object indexing to look up *User* objects by name executes in 130ms. But if $User[uc.actor.name]$ is replaced by the semantically equivalent $User{\rightarrow}select(name : uc.actor.name)$, the execution time increases to 72,644ms, an additional 73ms for each lookup.

Omit Negative Application Conditions removes the cost of evaluating the succedent of a constraint from each application attempt of the constraint: this can reduce the total transformation execution time by 50%. This improvement occurs if the cost of evaluating the antecedent $SCond$ is 0, while the cost of evaluating the succedent $Succ$ is equal to the cost of executing $stat(Succ)$, and all instances of the constraint are applied. In one test case of [33] involving a model of size 2000, and a constraint with a succedent conflicting with the antecedent, an improvement of execution time from 3s to 2s was observed by applying the pattern.

Replace Fixed-point by Bounded Iteration likewise can substantially reduce execution times by avoiding redundant application checks of constraint applications to newly-created elements that cannot satisfy the application conditions. Again, improvements of the order of 50% in execution time are possible. For example, if each application of the constraint creates a new instance of the source domain S_i, which does not satisfy $SCond$, and the cost of executing $Succ$ is 0, the optimization removes $card(S_i@pre)$ redundant evaluations of $SCond$.

Decompose Complex Navigations can reduce the cost of selecting each source domain element in the implementation of a constraint, from an evaluation cost of the form $N_1 * ... * N_k$ to a cost $N_1 + ... + N_k$. An example is the constraint test $a : specialization.specific.ownedAttribute$ in the refactoring example of [33]. By splitting the test into

$$s : specialization.specific \text{ and } a : s.ownedAttribute$$

the execution time is reduced from 472s to 273s, in an example with 500 subclasses and 10 attributes in each class [33].

Remove Duplicated Expression Evaluations modularizes specifications by increasing factorization: if there are n occurrences of an expression e in a specification, which can be factored out, then the total expression complexity of the specification is decreased by $n-1$ times the complexity of e. The execution time is potentially reduced by $n - 1$ times the evaluation time of e, but the costs of performing the caching of e's value also need to be taken into account. An example where the execution time is reduced is given in Section 6.7. Without the pattern, the cost of evaluating $c.rootClass()$ is proportional to the inheritance depth of c, each time it is evaluated. By precomputing each $rootClass$ value as an association, the cost of evaluating $c.rootClass$ becomes constant.

Table 4 shows the general effects of applying an architectural pattern on the maximum complexity of a transformation in a transformation system, and on the modularity, efficiency and memory usage of the transformations.

Pattern	*Primary benefit*	*Secondary benefit*
Phased Construction	Reduces average rule complexity	Increases modularity
Structure Preservation	As above	As above
Entity Splitting	As above	Increases cohesion
Entity Merging	As above	Increases modularity
Map Objects Before Links	As above	May improve efficiency
Parallel Composition	Increases cohesion	Reduces size
Auxiliary Metamodel	May reduce maximum expression complexity	Improves modularity
Construction and Cleanup	May reduce average rule complexity	Improves modularity
Recursive Descent	Controls rule application orders	Controls rule application scope
Replace Explicit by Implicit	Reduces explicit coupling	Improves modularity
Introduce Rule Inheritance	Reduces average rule complexity	Increases factoring
Unique Instantiation	Prevents duplicate object construction	Reduces memory usage
Object Indexing	Reduces time of object lookup	Reduces syntactic complexity
Omit Negative Application Conds.	Reduces execution time	Reduces implementation code size
Replace Fixed-point by Bounded Iteration	As above	As above
Decompose Complex Navigations	Reduces memory usage	Reduces execution time
Remove Duplicated Expression Evaluations	Reduces execution time	Improves factorization
Restrict Input Ranges	Reduces execution time	Reduces memory usage
Implicit Copy	Reduces overall size	Reduces overall complexity
Model Visitor	Improves modularity	Improves extensibility
Text Templates	Reduces transformation size	Reduces transformation complexity
Replace Abstract by Concrete Syn.	Reduces transformation size	Reduces transformation complexity
Simulating Universal Quantification	Permits more expressive rules	
Simulating Explicit Rule Scheduling	Permits greater execution control	

TABLE 3
Model transformation pattern effects

Pattern	Complexity *comp*	Cohesion *cohe*	Execution time	Memory usage
Phased Model Construction	Reduced from $comp(\tau_1) + comp(\tau_2)$ to $comp(\tau_1)$, $comp(\tau_2)$	Increased from $\frac{1}{N}$ to $\frac{1}{N1}$, $\frac{1}{N2}$ where $N = N1 + N2$	Reduced from $t_1 + t_2$ to t_1, t_2	Reduced input model sizes
Target Model Splitting	As above	As above	As above	Reduced output model sizes
Model Merging	As above	May not be increased	As above	Reduced input model sizes
Auxiliary Models	Reduced from $comp(\tau)$ to $comp(\tau_1)$, $comp(\tau_2)$	May not be increased	May increase	May increase
Filter Before Processing	Not changed, if $comp(filter) \leq comp(\tau)$	Not changed	Reduced for τ	Reduced for τ

TABLE 4
Effects of architectural patterns

Phased Model Construction typically divides the transformation specification into two non-empty parts, thus reducing the average complexity of the transformations. It will increase cohesion (of the subtransformations, compared to that of the original transformation) if the two subtransformations have disjoint write frames, which is the usual case. The execution costs and memory usage of each subtransformation should also be reduced compared to the original transformation. This is also the case for Model Merging and Target Model Splitting.

For Auxiliary Metamodels, the complexity of each subtransformation should be less than that of the original transformation. In one application of the pattern to a transformation which finds all pairs of intersecting 3-cycles of edges in a directed graph, by first finding all 3-cycles, then identifying intersecting pairs, the complexity of the original transformation was 53, and its cohesion 0.5, whilst the separated subtransformations had complexities 39 and 14, each with cohesion 1.

For Filter Before Processing, the execution time and resource usage of the main transformation should be reduced by introducing the filter. Applying a pre-filtering to the IMDb movie data of [25] leads to substantial reductions in data size: the 44931 persons in the first data set are reduced to 2809 by the filter, and the 5000 movies to 1366. This leads to a reduction in execution time (for identifying couples of actors working together in at least 3 movies) from 890s without the filter, to 6s with the filter [51].

15 Related Work

In [57] the concept of automated derivation of transformation implementations from specifications written in a constructive type theory was introduced. A single undecomposed constraint of nested $\forall \Rightarrow \exists$ form defines the entire transformation, and the constructive proof of this formula produces a function which implements the transformation. In [58] the $\exists x.P$ quantifier is interpreted as an obligation to construct a witness element x satisfying P. A partial ordering of entity types in the source and target languages is used to successively construct such witnesses. The synthesized implementations are structured using the Recursive Descent pattern.

In [8], a transformation specification pattern is introduced, *Transformation parameters*, to represent the case where some auxiliary information is needed to configure a transformation. This is a special case of the Auxiliary Metamodel pattern.

In [11], specific patterns for the optimization of rule executions are defined: *short-circuit boolean expressions evaluation*, to organize conditional tests to avoid redundant computation; *collections*, to guide the choice of efficient OCL collections; *usage of iterators*, to optimize OCL iterator use; *finding constant expressions*, to precompute expressions within rules which have the same value in every rule application. Of these, the fourth can be considered as a special case of Remove Duplicated Expression Evaluations.

In [14] three general styles for transformations are described: source-driven transformations, target-driven transformations, and aspect-driven transformations. The first two correspond to Entity Splitting and Entity Merging in our terminology, the third corresponds to our Parallel Composition of transformations, where different aspects of the target model are constructed using separate parallel transformations.

Patterns specific to graph transformation languages are defined in [1]: *leaf collector pattern*, to characterize the common processing pattern which accumulates all instances which are leaves with respect to some hierarchical relationship; *map-using-link pattern*, a technique for constructing a trace; *transitive closure pattern*, to characterize the common processing pattern which constructs the transitive closure of some relation; *proxy generator idiom*. Of these, map-using-link is a special case of the Auxiliary Metamodel pattern. Computation of a transitive closure is perhaps best formalized as a generic transformation, as in Viatra [68]. In [26] five model transformation design patterns are described: *Mapping*, *Refinement*, *Abstraction*, which can be considered as general categories of model transformation (the cases of 1-to-1, 1-to-many, and many-to-1 relations between the source and target model elements, respectively) rather than as specific design patterns, *Duality*, for computing the edge-node dual of a software engineering model, and *Flattening*, for contracting a source model object hierarchy. The latter two are also candidates for formalization as generic transformations. In [29] four patterns are defined: *Element mapping*, *Element mapping with variability*, which are special cases of Entity Splitting, and *Attribute mapping* and *Link mapping*, which are elementary aspects of transformations specified with the Phased Construction form (the derivation of target model instance attribute values from those of source model instance attribute values, and the derivation of target model links from those of the source model, respectively).

Other possible patterns for model transformations can be identified from published examples of model transformations: *move constant expressions out of loops* and *inlining operation calls*. These are general code optimization strategies, not specific to model transformations.

Model transformation patterns are distinguished both from the transformation primitives defined by [29], [67] and generic model transformations [17], [69], which provide a specification of a family of transformations, parameterized upon the source and target model entities and features they operate on. As we suggested above, very specific 'patterns' such as the computation of a transitive closure, are more appropriately defined as generic transformations. Patterns are more open-ended and diverse in their instantiations, in contrast to the fixed templates characteristic of generic transformations.

Patterns which concern the optimization of the generated code of a transformation, such as moving constant expressions out of loops, or inlining operation calls, are akin to conventional compiler optimizations and can in principle be automated by a transformation development tool, so that these also can be distinguished from patterns that require human expertise and creativity to apply, such as the Auxiliary Metamodel pattern.

It would be interesting to empirically evaluate the benefits of transformation patterns for developers and readers of transformation specifications. Such studies have been conducted for other families of patterns with generally positive evidence for improved program comprehension [60]. We have identified in [45] that the use of model transformation patterns can significantly simplify transformation verification.

16 Conclusion

We have described 24 model transformation specification and design patterns, and 5 model transformation architecture patterns. We have identified that transformation patterns can themselves be regarded as meta-transformations, in that the application of a pattern to a transformation restructures the transformation to improve some quality characteristic such as modularity or efficiency. Thus, one natural way to implement the application of model transformation patterns is by means of higher-order transformation facilities, where these are provided by a transformation language or tool.

We have focused on patterns for refinement, abstraction, migration and refactoring transformations. Further work is needed to identify patterns in other areas, such as model comparison, merging and transformation bidirectionality. Further development of our measurement framework could also be useful, to provide automated identification of specification and design flaws in transformations.

Acknowledgment

The work presented here was carried out in the EPSRC HoRTMoDA project at King's College London.

References

[1] A. Agrawal, A. Vizhanyo, Z. Kalmar, F. Shi, A. Narayanan, G. Karsai, *Reusable Idioms and Patterns in Graph Transformation Languages*, Electronic notes in Theoretical Computer Science, pp. 181–192, 2005.

[2] D. Akehurst, W. Howells, K. McDonald-Maier, *Kent Model Transformation Language*, Model Transformations in Practice Workshop, MODELS 2005, Montego Bay, 2005, http://kar.kent.ac.uk/id/eprint/8905.

[3] C. Alexander, *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[4] M. van Amstel, S. Bosems, I. Kurtev, L. F. Pires, *Performance in Model Transformations: experiments with ATL and QVT*, ICMT 2011, LNCS 6707, pp. 198–212, 2011.

[5] K. Anastasakis, B. Bordbar, G. Georg, I. Ray, *On challenges of model transformation from UML to Alloy*, SoSyM, vol. 9, no. 1, 2010, pp. 69–86.

[6] D. Balasubramanian, A. Narayanan, C. van Buskirk, G. Karsai, *The Graph Rewriting and Transformation Language: GReAT*, GraBaTs 2006, ECAESST vol. 1, 2006.

[7] I. Bayley, H. Zhu, *On the composition of design patterns*, QSIC 2008, IEEE Computer Society, 2008, pp. 27–36.

[8] J. Bezivin, F. Jouault, J. Palies, *Towards Model Transformation Design Patterns*, ATLAS group, University of Nantes, 2003.

[9] J. Cabot, R. Clariso, E. Guerra, J. De Lara, *Verification and Validation of Declarative Model-to-Model Transformations Through Invariants*, Journal of Systems and Software, 83(2), 2010, pp. 283–302.

[10] C. Cleaveland, *Program Generators with XML and Java*, Prentice Hall, 2001.

[11] J. S. Cuadrado, F. Jouault, J. G. Molina, J. Bezivin, *Optimization patterns for OCL-based model transformations*, MODELS 2008, vol. 5421 LNCS, Springer-Verlag, pp. 273–284, 2008.

[12] J. Cuadrado, J. Molina, *Modularization of model transformations through a phasing mechanism*, Software Systems Modeling, Vol. 8, No. 3, pp. 325–345, 2009.

[13] K. Czarnecki, S. Helsen, *Feature-based survey of model transformation approaches*, IBM Systems Journal, vol. 45, no. 3, 2006, pp. 621–645.

[14] K. Duddy, A. Gerber, M. Lawley, K. Raymond, J. Steel, *Model transformation: a declarative, reusable pattern approach*, 7th International Enterprise Distributed Object Computing Conference (EDOC '03), 2003, pp. 174–185.

[15] Eclipsepedia, *ATL User Guide*, http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language, 2014.

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.

[17] R. France, S. Chosh, E. Song, D. Kim, *A metamodelling approach to pattern-based model refactoring*, IEEE Software, vol. 20, no. 5, pp. 52–58, 2003.

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

[19] M. Grand, *Patterns in Java*, Volume 1, Wiley, 1998.

[20] T. Goldschmidt, G. Wachsmuth, *Refinement transformation support for QVT relational transformations*, FZI, Karlsruhe, Germany, 2011.

[21] P. Van Gorp, S. Mazanek, A. Rensink, *Live challenge problem*, TTC 2010, Malaga, July 2010. http://is.tm.tue.nl/staff/pvgorp/events/TTC2010/?page=CaseStudyDescriptions08.

[22] R. Gronmo, B. Moller-Pedersen, G. K. Olsen, *Comparison of Three Model Transformation Languages*, proceedings of ECMDA-FA, LLNCS 5562, pp. 2–17, 2009.

[23] E. Guerra, J. de Lara, D. Kolovos, R. Paige, O. Marchi dos Santos, *transML: A family of languages to model model transformations*, MODELS 2010, LNCS vol. 6394, Springer-Verlag, 2010.

[24] M. Hafiz, Security Pattern Catalog, www.munawarhafiz.com/securitypatterncatalog/index.php, 2013.

[25] T. Horn, C. Krause, M. Tichy, *The TTC 2014 Movie Database Case*, TTC 2014.

[26] M. E. Iacob, M. W. A. Steen, L. Heerink, *Reusable model transformation patterns*, Enterprise Distributed Object Computing Conference Workshops, 2008, pp. 1–10, doi:10.1109/EDOCW.2008.51.

[27] E. Jakumeit, S. Buchwald, M. Kroll, *GrGen.NET: the expressive, convenient and fast graph rewrite system*, International Journal on Software Tools for Technology Transfer (STTT), 12: 263–271, 2010.

[28] Jamda, http://jamda.sourceforge.net, 2012.

[29] J. Johannes, S. Zschaler, M. Fernandez, A. Castillo, D. Kolovos, R. Paige, *Abstracting complex languages through transformation and composition*, MODELS 2009, LNCS 5795, pp. 546–550, 2009.

[30] F. Jouault, I. Kurtev, *Transforming Models with ATL*, in MoDELS 2005, LNCS Vol. 3844, pp. 128–138, Springer-Verlag, 2006.

[31] J. Kerievsky, *Refactoring to Patterns*, Addison Wesley, 2004.

[32] Kermeta, http://www.kermeta.org, 2010.

[33] S. Kolahdouz-Rahimi, K. Lano, S. Pillay, J. Troya, P. Van Gorp, *Evaluation of model transformation approaches for model refactoring*, Science of Computer Programming, 2013, http://dx.doi.org/10.1016/j.scico.2013.07.013.

[34] D. Kolovos, R. Paige, F. Polack, *The Epsilon Transformation Language*, in ICMT 2008, LNCS Vol. 5063, pp. 46–60, Springer-Verlag, 2008.

[35] I. Kurtev, K. Van den Berg, F. Joualt, *Rule-based modularization in model transformation languages illustrated with ATL*, Proceedings 2006 ACM Symposium on Applied Computing (SAC 06), ACM Press, pp. 1202–1209, 2006.

[36] K. Lano, J. Bicarregui, *Semantics and Transformations for UML Models*, UML 98, Mulhouse, France, June 1998, Springer-Verlag LNCS vol. 1618, 1998, pp. 107–119.

[37] K. Lano, *Model-driven software development with UML and Java*, Cengage, London, 2009.

[38] K. Lano, *A catalogue of UML model transformations*, http://www.dcs.kcl.ac.uk/staff/kcl/tcat.pdf, 2006.

[39] K. Lano, S. Kolahdouz-Rahimi, *Slicing of UML models using Model Transformations*, MODELS 2010, LNCS vol. 6395, pp. 228–242, 2010.

[40] K. Lano, S. Kolahdouz-Rahimi, *Migration case study using UML-RSDS*, TTC 2010, Malaga, Spain, July 2010. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/submissions/final/uml-rsds.pdf.

[41] K. Lano, S. Kolahdouz-Rahimi, *Model-driven development of model transformations*, ICMT 2011, June 2011. LNCS vol. 6707, 2011, pp. 47–61.

[42] K. Lano, S. Kolahdouz-Rahimi, *Solving the TTC 2011 migration case study with UML-RSDS*, TTC 2011, EPTCS vol. 74, pp. 36–41, 2011.

[43] K. Lano, S. Kolahdouz-Rahimi, *Design patterns for model transformations*, ICSEA 2011, IARIA, 2011, pp. 263–268.

[44] K. Lano, S. Kolahdouz-Rahimi, *Model Transformation Specification and Design*, in 'Advances in Computers', Vol 85. Burlington: Academic Press, 2012, pp. 123–164.

[45] K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Comparing verification techniques for model transformations*, Modevva workshop, MODELS 2012, ACM Press, pp. 23–28.

[46] K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, J. Terrell, S. Zschaler, *Correct-by-construction synthesis of model transformations using design patterns*, SoSyM, vol. 13, no. 2, May 2014, pp. 412–436.

[47] K. Lano, *The UML-RSDS Manual*, www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrsds.pdf, 2014.

[48] K. Lano, S. Kolahdouz-Rahimi, *Optimizing Model-transformations using Design Patterns*, MODELSWARD 2013, http://www.modelsward.org/.

[49] K. Lano, S. Kolahdouz-Rahimi, K. Maroukian, *Solving the Petri-Nets to Statecharts Transformation Case with UML-RSDS*, TTC 2013, EPTCS, 2013.

[50] K. Lano, S. Kolahdouz-Rahimi, *Case study: Class diagram restructuring*, www.planet-sl.org/community/_/ttc/ttc2013/cases/ClassDiagramRestructuring, 2013.

[51] K. Lano, S. Yassipour-Tehrani, *Solving the Movie Database Case with UML-RSDS*, TTC 2014.

[52] D. Michie, *Memo Functions and Machine Learning*, Nature, vol. 218, pp. 19–22, 1968.

[53] OMG, *Query/View/Transformation Specification*, ptc/05-11-01, 2005.

[54] OMG, *Query/View/Transformation Specification*, annex A, 2010.

[55] Openarchitectureware, http://www.openarchitectureware.org, 2012.

[56] OptXware, *The Viatra-I Model Transformation Framework Users Guide*, 2010.

[57] I. Poernomo, *Proofs as model transformations*, ICMT 2008, LNCS vol. 5063, 2008, pp. 214–228.

[58] I. Poernomo, J. Terrell, *Correct-by-construction Model Transformations from partially-ordered specifications in Coq*, ICFEM 2010, LNCS vol. 6447, 2010, pp. 56–73.

[59] C. Pons, R. Giandini, G. Perez, G. Baum, *A two-level calculus for composing hybrid QVT transformations*, SCCC 2009, IEEE Press, pp. 105–114.

[60] L. Prechelt, *An experiment on the usefulness of design patterns*, Informatics Faculty, University of Karlsruhe, Germany, 1997.

[61] A. Rensink, J-H. Kuperus, *Repotting the Geraniums: on nested graph transformation rules*, proceedings of GT-VMT 2009, Electronic communications of the EASST vol 18, 2009, http://dblp.uni-trier.de/db/journals/eceasst/eceasst18.html#RensinkK09.

[62] R. Romeikat, S. Roser, P. Mullender, B. Bauer, *Translation of QVT Relations into QVT Operational Mappings*, ICMT 2008, Springer-Verlag, pp. 137–151.

[63] L. Rose, D. Kolovos, R. Paige, F. Polack, *Migrating activity diagrams with Epsilon Flock*, TTC 2010, pp. 184–198.

[64] L. Rose, M. Herrmannsdoerfer, S. Mazanek et al., *Graph and Model Transformation Tools for Model Migration*, Software and Systems Modeling, April 2012, doi:10.1007/s10270-012-0245-0.

[65] A. Schurr, *Specification of graph translators with triple graph grammars*, WG '94, LNCS vol. 903, Springer, 1994, pp. 151–163.

[66] B. Selic, *What will it take? A view on adoption of model-based methods in practice*, Software systems modeling, 11: 513–526, 2012.

[67] E. Syriani, H. Vangheluwe, *De-/re-constructing model transformation languages*, Proceedings of 9th international workshop GT-VMT, Electronic Communications of EASST 29, 2010, http://journal.ub.tu-berlin.de/eceasst/article/view/407.

[68] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovsky, U. Prange, D. Varro, S. Varro-Gyapay, *Model transformation by graph transformation: a comparative study*, MODELS 2005.

[69] M. Tisi, J. Cabot, F. Jouault, *Improving higher-order transformations support in ATL*, ICMT 2010, LNCS vol. 6142, pp. 215–229, 2010.

[70] P. Van Gorp, L. Rose, *The Petri-Nets to Statecharts Transformation Case*, TTC 2013, EPTCS, 2013, pp. 16–31.

[71] D. Varro, M. Asztalos, D. Bisztray, A. Boronat, D-H. Dang, R. Geis, J. Greenyer, P. Van Gorp, O. Kniemeyer, A. Narayanan, E. Rencis, E. Weinell, *Transformation of UML models to CSP: a case study for graph transformation tools*, AGTIVE 2007, LNCS 5088, pp. 540–565, 2007.

[72] S. Zschaler, I. Poernomo, J. Terrell, *Towards using constructive type theory for verifiable modular transformations*, presented at FREECO 2011, 2011.