Patterns for Model Transformation Specification and Implementation

K. Lano, S. Kolahdouz-Rahimi, I. Poernomo, J. Terrell
Dept. of Informatics, King’s College London

Abstract. In this paper we describe standard structures for model transformation specifications and implementations, which serve as patterns for constructing a wide range of model transformations. We use specification patterns to derive relationships between invertibility, change-propagation and the existence of language-level interpretations for transformations that are defined using the pattern. We also consider how these specification patterns can be used to systematically construct implementations for the specifications, and what software patterns (both variations on well-known design patterns, and those specific to the model transformation domain) are particularly relevant to model transformation implementation.

1 Introduction

Model transformations are an essential part of development approaches such as Model-driven Architecture (MDA) [25] and Model-driven Development (MDD). In this paper we will specifically consider model transformations between languages specified as metamodels using the UML-RSDS subset of UML [14].

Model-transformations are becoming large and complex and business-critical systems in their own right, and so require systematic development. The concept of software pattern [7] has been used to improve the reusability and flexibility of general software systems, and can also be applied to improve model-transformation construction.

The aim of this paper is to describe typical patterns for the specification of model transformations, to apply these patterns to deduce properties of the model transformations, such as invertibility, and to use them to support the automated derivation of (correct by construction) implementations of transformations. This process has been implemented as part of the UML-RSDS toolset for MDD using UML. It could also be applied to other model transformation approaches, such as QVT [23] or ATL [9]. The patterns described here have been derived from experimentation with a wide range of different transformations, and from published examples of transformations.

In Section 2 we define a semantics for model transformations, and a framework for the development and verification of model transformations. Section 3 describes typical patterns for the specification of model transformations, and Section 4 describes typical patterns for the implementation of model transformations, and strategies for the derivation of implementations from specifications.
Section 5 describes a metalevel model transformation which maps model transformation specifications to designs. Appendix A summarises the patterns, and Appendix B defines read and write frames.

2 Model Transformation Semantics

We will use the four-level metamodelling framework of UML as the context for transformations [26]. This consists of levels M0: models which consist of run-time instances of M1 models, which are user models such as class diagrams, which are in turn instances of metamodels M2, such as the definition of the class diagram language itself. Level M3 contains the EMOF and CMOF languages for defining metamodels. Other languages such as EMF Ecore could alternatively be used at level M3 [5].

Most transformations operate upon M1 level models, so we will refer to the M2 level as the language level (the models at this level define the languages which the transformation relates) and to the M1 level as the model level (the models which the transformation operates upon). We will usually specify transformations as constraints at the M2 level, and they will be implemented as operations at this level also.

For each model $M$ at levels M2 and M3, we can define (i) a logical language $\mathcal{L}_M$ that corresponds to $M$, and (ii) a logical theory $\Gamma_M$ in $\mathcal{L}_M$, which defines the semantic meaning of $M$, including any internal constraints of $M$ [18]. The set-theory based axiomatic semantics of UML described in Chapter 6 of [13] is used to define $\Gamma_M$ and $\mathcal{L}_M$. If $M$ at level M1 is itself a UML-based model to which a semantics $\Gamma_M$ can be assigned, then also $\Gamma_M$ and $\mathcal{L}_M$ will be defined for $M$.

$\mathcal{L}_M$ consists of type symbols for each type defined in $M$, including primitive types such as integers, reals, booleans and strings which are normally included in models, and each entity type defined in $M$. The boolean operators and, implies, forAll, exists of OCL are semantically interpreted by the logical connectives $\land$, $\Rightarrow$, $\forall$, $\exists$. For each entity $C$ of $M$ there are attribute symbols $att(c : C) : T$ for each property $att$ of type $T$ in the feature set of $C$, and action symbols $op(c : C, p : P)$ for each operation $op(p : P)$ in the features of $C$. There are attributes $\bar{C}$ to denote the set of instances of each entity $C$ (corresponding to $C.allInstances()$ in OCL). Collection types and operations on these and the primitive types are also usually included.

$\Gamma_M$ includes axioms expressing the multiplicities of association ends, the mutual inverse property of opposite association ends, deletion propagation through composite aggregations, the existence of generalisation relations, and the logical semantics of any explicit constraints in $M$. Constraints of $M$ are also expressed as axioms in $\Gamma_M$.

For a sentence $\varphi$ in $\mathcal{L}_M$, there is the usual notion of logical consequence:

\[ \Gamma_M \vdash \varphi \]
means the sentence is provable from the theory of $M$, and so holds in $M$. We may specify inference within a particular language $L$ by the notation $\vdash_L$ if $L$ is not clear from the context.

If $M$ is at the M1 level and is an instance of a language $L$ at the M2 level, then it satisfies all the properties of $\Gamma_L$, although these may not be expressible within $L_M$ itself. We use the notation $M \models \varphi$ to express satisfaction of an $L_L$ sentence $\varphi$ in $M$.

The collection of M1 level models of an M2 model $L$ is $\text{Models}_L$. We may simply write $m : L$ to mean $m \in \text{Models}_L$. In concrete terms a model can be considered to be a tuple

$$(E_1, \ldots, E_n, f_1, \ldots, f_m)$$

of the sets $E_i$ of instances of each entity $E_i$ of $L$, and of the maps $f_j : E_i \rightarrow T$ representing the values of data features of these entities.

A model transformation $\tau$ from a language $L_1$ to a language $L_2$ can be considered to define a relationship between the model sets of these languages:

$$\text{Rel}_\tau : \text{Models}_{L_1} \leftrightarrow \text{Models}_{L_2}$$

A model transformation $\tau$ is invertible if there is an inverse transformation $\sigma$ from $L_2$ to $L_1$ such that $\tau$ followed by $\sigma$ is the identity relation on $\text{Models}_{L_1}$:

$$\text{Rel}_\tau \cdot \text{Rel}_\sigma = \text{id}_{\text{Models}_{L_1}}$$

A transformation is change-propagating if changes $\Delta s$ of a source model $s$ can be used to compute changes $\Delta t$ to a target model, without the need to reapply the transformation to the entire new model $s + \Delta s$ [30].

2.1 Model transformation correctness

By using the above semantics for transformations, we can precisely define correctness criteria for a model transformation $\tau$ from a language $L_1$ to a language $L_2$ [11]:

**Syntactic correctness** For each model which conforms to (is a model of the language) $L_1$, and to which the transformation can be applied, the transformed model conforms to $L_2$:

$$\forall M_1 : L_1; M_2 : \text{Rel}_\tau(M_1, M_2) \Rightarrow M_2 : L_2$$

**Semantic correctness** At the language level this means that for each property of the source model which should be preserved (correctness properties), the target model satisfies the property, under a fixed interpretation $\chi$ of the source language into the target language. There may also be model-level properties which should be preserved via a model-level interpretation $\zeta$. 
1. (Language-level correctness): each language-level property \( \varphi : \mathcal{L}_{L1} \) satisfied by a source model \( M_1 \) is also satisfied, under an interpretation \( \chi \) on language-level expressions, in \( M_2 \):

\[
\forall M_1 : L1; \ M_2 : L2 \cdot \text{Rel}_\tau(M_1, M_2) \land M_1 \models \varphi \Rightarrow M_2 \models \chi(\varphi)
\]

This shows that \( L2 \) is at least as expressive as \( L1 \), and that users of \( M_2 \) can view it as a model of \( L1 \), because the data of \( M_1 \) can be expressed in terms of that of \( M_2 \). In particular, no information about \( M_1 \) is lost by \( \tau \).

2. (Model-level correctness): each model-level property \( \varphi : \mathcal{L}_{M_1} \) of a source model \( M_1 \) is also true, under an interpretation \( \zeta \) on model-level expressions, in \( M_2 \):

\[
\forall M_1 : L1; \ M_2 : L2 \cdot \text{Rel}_\tau(M_1, M_2) \land \Gamma_{M_1} \vdash \varphi \Rightarrow \Gamma_{M_2} \vdash \zeta(\varphi)
\]

for \( \varphi \in \Gamma_{M_1} \).

This means that internal constraints of \( M_1 \) remain valid in interpreted form in \( M_2 \), for example, that subclassing in a UML model is mapped to subsetting of sets of table primary key values, in a transformation from UML to relational databases [24].

The first property is termed \textit{Strong executability} in [3].

Two key properties of a transformation implementation are \textit{confluence} and \textit{definedness}:

- Confluence means that the order of execution of transformation rules within a transformation does not affect the result, provided the rules are executed in an order permitted by the transformation.
- Definedness means that the implementation is well-defined on the set of input models. In particular, recursions and iterations are terminating, exceptions from the evaluation of undefined expressions should not occur, and rules should only be applied if their preconditions hold.

### 2.2 Development process for model transformations

In this section we outline a general model-driven development process for model transformations specified as constraints and operations in UML. We assume that the source and target metamodels of a transformation are specified as UML-RSDS class diagrams\(^1\), \( S \) and \( T \), respectively, possibly with OCL constraints defining semantic properties of these languages.

For a transformation \( \tau \) from \( S \) to \( T \), there are four separate predicates which characterise its global properties, and which need to be considered in its specification and design [18]:

\(^1\) Essentially these are UML class diagrams without packages, multiple inheritance, association classes, qualified associations, n-ary associations (n > 2) or non-leaf concrete classes.
1. \textit{Asm} – assumptions, expressed in $\mathcal{L}_{S\cup T}$, which can be assumed to be true before the transformation is applied. These may be assertions that the source model is syntactically correct, that the target model is empty, or more specialised assumptions necessary for $\tau$ to be well-defined. These are preconditions of the use case of the transformation.

2. \textit{Ens} – properties, usually expressed in $\mathcal{L}_T$, which the transformation should ensure about the target model at termination of the transformation. These properties usually include the constraints of $T$, in order that syntactic correctness holds. For update-in-place transformations, \textit{Ens} may refer to the pre-state versions of model data.

3. \textit{Pres} – properties, usually expressed in $\mathcal{L}_S$, which the transformation should preserve from the source model to the target model, under a language-level interpretation $\chi$. This could include language-level semantic correctness. Properties at the model level may also be specified for preservation via a model-level interpretation $\zeta$.

4. \textit{Cons} – constraints, expressed in $\mathcal{L}_{S\cup T}$, which define the transformation as a relationship between the elements of the source and target models, which should hold at termination of the transformation. Update-in-place transformations can be specified by using a syntactically distinct copy of the source language, for example by postfixing all its entity and feature names by $\@\text{pre}$.

\textit{Cons} corresponds to the postconditions of the use case of the transformation.

We can express these predicates using OCL notation, this corresponds directly to a fully formal version in the axiomatic UML semantics. Together these predicates give a global and declarative definition of the transformation and its requirements, so that the correctness of a transformation may be analysed at the specification level, independently of how it is implemented.

The following should be provable:

\[ \text{Cons}, \Gamma_S \vdash_{\mathcal{L}_{S\cup T}} \text{Ens} \]

We assume the source language theory holds for the source model, generally this should be confirmed by some checking transformation before applying the mapping transformation.

Likewise, \textit{Cons} should prove that \textit{Pres} are preserved, via a suitable interpretation $\chi$ from the source language to the target language:

\[ \text{Cons}, \text{Pres}, \Gamma_S \vdash_{\mathcal{L}_S\cup\mathcal{T}} \chi(\text{Pres}) \]

Development of the transformation then involves the construction of a design which ensures that the relationship \textit{Cons} holds between the source and target models. This may involve decomposing the transformation into phases or sub-transformations, each with their own specifications. By reasoning using the weakest-precondition operator $[ ]$ the composition of phases should be shown to achieve \textit{Cons}:

\[ \Gamma_S \vdash_{\mathcal{L}_{S\cup T}} \text{Asm} \Rightarrow [\text{activity}]\text{Cons} \]
where \textit{activity} is the algorithm of the transformation. Each statement form of the statement language (Figure 3) has a corresponding definition of [\[\]].

In many cases, the derivation of a correct-by-construction design from the specification can be automated, as we describe in Sections 4.4 and 5. Executable code can then be automatically generated from the design.

3 Specification Patterns

In this section we describe characteristic structures for model transformation specification, and the consequences of these structures for the invertibility, change-propagation and existence of interpretations for the transformation.

3.1 Categories of model transformation

Model transformations can be classified in different ways \cite{4,12}. At a syntactic level, we can differentiate between those transformations where the source and target languages \(S\) and \(T\) are entirely disjoint, or where they overlap, or where one is a sub-language of the other. In the final case, transformations may be \textit{update-in-place}, ie, they modify the elements of an existing model, rather than creating an entirely new model.

Semantically, a transformation can be classified in terms of its role in a development process:

\textbf{Refinement} A transformation that replaces source model elements by more complex target elements or structures to map an abstract model (such as a PIM) to a more specific version (such as a PSM). Code generation can be considered as a specific case.

\textbf{Abstraction} A transformation that provides an abstraction of a model, such as the result of a query over the model. This is the opposite of a refinement.

\textbf{Quality improvement} A transformation that remains at the same abstraction level, but that re-organises a model to achieve some quality goal (eg, removing duplicated attributes from a class diagram). Usually update-in-place.

\textbf{Re-expression} A transformation that maps a model in one language into its equivalent in another language at the same level of abstraction, eg, migration transformations from one version of a language to another version.

A small example of a re-expression transformation is a mapping between trees and graphs. Figure 1 shows the source and target metamodels of the transformation. The tree metamodel has the language constraint \(Asm1\) that there are no non-trivial cycles in the \textit{parent} relationship:

\[ t : Tree \text{ and } t \neq t.\text{parent} \implies t \notin t.\text{parent}^+ \]

where \(r^+\) is the non-reflexive transitive closure of \(r\). Trees may be their own parent if they are the root node of a tree. In mapping from trees to graphs we also assume that the graph model is empty (\(Asm2\)): \(Edge = \{\}\) and \(Node = \{\}\).

The graph metamodel has the constraint that edges must always connect different nodes (Ens1):

\[ e : \text{Edge implies } e\.source \neq e\.target \]

and that edges are uniquely defined by their source and target, together (Ens2):

\[ e1 : \text{Edge and } e2 : \text{Edge and} \]
\[ e1\.source = e2\.source \text{ and } e1\.target = e2\.target \text{ implies } e1 = e2 \]

![Diagram of Tree to graph transformation metamodels](image)

Fig. 1. Tree to graph transformation metamodels

The *identity* constraint means that tree nodes must have unique names, and likewise for graph nodes.

The transformation relates tree objects in the source model to node objects in the target model with the same name, and defines that there is an edge object in the target model for each non-trivial relationship from a tree node to its parent.

### 3.2 Conjunctive-implicative form

We consider first the case of unidirectional transformations \( \tau \), mapping from one language \( S \) to another (disjoint) language \( T \), and with separate source and target models where the target model is empty at the start of the transformation and the source model is not changed by the transformation, ie, not update-in-place transformations. Such transformations include many cases of re-expression transformations such as model migrations, and refinements and abstractions.

Let \( S_1, ..., S_n \) be the entities of \( S \) which are relevant to the transformation. Then one common specification pattern [28] is that \( Cons \) will have the general form of a conjunction of clauses, each of the form

\[ \forall s : S_i \cdot SCond_{i,j} \text{ implies } \exists t : T_{i,j} \cdot TCond_{i,j} \text{ and } Post_{i,j} \]

where \( SCond_{i,j} \) is a predicate over the source model elements only, \( T_{i,j} \) is some entity of \( T \), \( TCond_{i,j} \) is a condition in \( T \) elements only, eg., to specify explicit values for \( t \)'s attributes, and \( Post_{i,j} \) refers to both \( t \) and \( s \) to specify \( t \)'s attributes
and possibly linked (dependent) objects in terms of \( s \)'s attributes and linked objects. \( SCond \) and \( TCond \) do not contain quantifiers, \( Post \) may contain \( \exists \) quantifiers to specify creation/lookup of subordinate elements of \( t \). If the \( t \) should be unique for a given \( s \), the \( \exists_1 \) quantifier may be alternatively used in the succedent of clauses. Additional \( \forall \)-quantifiers may be used at the outer level of the constraint, if quantification over groups of source model elements is necessary, instead of over single elements.

For each \( S_i \) there may be several clauses, and the \( SCond_{i,j} \) should therefore be logically disjoint for distinct \( j \) if the corresponding right-hand sides of the implications are exclusive, in order that \( Cons \) is consistent.

If some \( S_i \) is a non-leaf class then its clauses can be duplicated for each of its subclasses, and the clauses for \( S_i \) itself removed (for abstract non-leaf classes). We will therefore consider only leaf \( S_i \) in this pattern.

The pattern typically applies when \( S \) and \( T \) are similar in structure, for example in the UML to relational database mapping of [24, 18], the source structure of Package, Class, Attribute corresponds to the target structure of Schema, Table, Column.

A simple example of the pattern is the specification of the tree to graph transformation by two global constraints:

**C1** “For each tree node in the source model there is a graph node in the target model with the same name”:

\[
\forall t : \text{Tree} \cdot \exists n : \text{Node} \cdot n.\text{name} = t.\text{name}
\]

**C2** “For each non-trivial parent relationship in the source model, there is a unique edge representing the relationship in the target model”:

\[
\forall t : \text{Tree} \cdot t.\text{parent} \neq t \text{ implies } \\
\exists_1 e : \text{Edge} \cdot e.\text{source} = \text{Node}[t.\text{name}] \text{ and } e.\text{target} = \text{Node}[t.\text{parent}.\text{name}]
\]

This illustrates a case where one source model element may produce two connected target model elements. The notation \( E[v] \) denotes the \( E \) instance with primary key value \( v \), if \( v \) is a single element, or the set of \( E \) instances with primary key values in \( v \), if \( v \) is a collection.

The constraints have a dual aspect: they express what conditions should be true at the completion of an entire transformation, but they can also be interpreted as the definitions of specific rules executed within the transformation. \( C1 \) can be operationally interpreted as “For each tree node in the source model, create a graph node in the target model with the same name” and \( C2 \) as “For each tree node with a distinct parent, create an edge linking the graph node corresponding to the tree node to the graph node corresponding to its parent, unless such an edge already exists.”

We say that a predicate \( P \) is localised to elements \( s_1 : S_{i1}, \ldots, s_p : S_{ip} \) in a language \( S \) if \( P \) only refers to expressions which are direct navigations from the \( s_i \), or are constants from primitive and enumeration data types. Ideally the \( SCond \) and \( TCond \) expressions in a constraint are localised to \( s \) and \( t \) respectively, and \( Post \) to \( s, t \).
Provided that the collection of $TCond_{i,j}$ conditions for each specific target entity $T_i = T_{i,j}$ are logically exclusive, we can deduce reverse constraints/rules from $Cons$, which express that elements of $T$ can only be created as a result of the application of one of the forward constraints/rules: each reverse rule has the form:

$$\forall t : T_{i,j} \cdot TCond_{i,j} \implies \exists s : S_i \cdot SCond_{i,j} \text{ and } Post'_{i,j}$$

where $Post'_{i,j}$ expresses the inverse of $Post_{i,j}$. This predicate may not be explicit as a definition of $s$ in terms of $t$, for example if $Post_{i,j}$ is

$$t.name = s.forename + " " + s.surname$$

there is no explicit inverse and $Post'_{i,j}$ is the same as $Post_{i,j}$.

The reverse rules for the tree to graph specification are:

**C3** “For each graph node in the target model there is a tree node in the source model with the same name”:

$$\forall g : Node \cdot \exists t : Tree \cdot t.name = g.name$$

**C4** “For each edge in the target model, there is a unique non-trivial parent relationship in the source model, which the edge represents”:

$$\forall e : Edge \cdot \exists t : Tree \cdot t.parent \neq t \text{ and } t = Tree[e.source.name] \text{ and } t.parent = Tree[e.target.name]$$

$C3$ and $C4$ are duals of $C1$ and $C2$. $C4$ could be simplified to the equivalent constraint:

$$\forall e : Edge \cdot Tree[e.source.name].parent = Tree[e.target.name]$$

Examples of transformations which fit this pattern are the model migration transformation of UML 1.4 state machines to UML 2.2 activity diagrams [17], and the QVT-R implementation of the UML to relational database mapping, with persistent classes mapping to tables, packages to schemas, attributes to columns, etc [18]. Likewise for the re-expression transformation example of [31].

From the pattern we can deduce some general properties of this form of transformation:

- Provided that the $SCond$ predicates are localised to $s$, the $TCond$ to $t$, and the $Post$ are localised to $s$ and $t$, all conditions can be effectively computed, and that the $Post$ are explicit, the transformation is change-propagating for creation of new $S_i$ elements: the corresponding new $T_{i,j}$ element(s) can be created and their feature values set according to which constraint has a true $SCond_{i,j}$ condition.
- Likewise, if these conditions hold, and the $Post'$ are localised, explicit and computable, we can change-propagate creation of new $T_{i,j}$ elements using the reverse rules.
– Change-propagation of deletion of $S_i$ and $T_{i,j}$ also follow under these assumptions, by considering the contrapositives of the reverse and forward rules, respectively.

General propagation of attribute value changes however requires some tracing mechanism (such as identity attributes or attached stereotypes of target model elements) to connect specific source elements to the target elements whose values depend upon them. In particular, a change of attribute values of some $s : S_i$ may alter which $SCond_{i,j}$ holds true, implying that the $T$ elements originally derived from $s$ should be deleted and new elements created.

The reverse rules of a transformation specification may provide only a partial inverse to the transformation, if it involves abstraction, because:

– Not all entities of the source language may be used by the transformation, so that a source model with instances of such entities cannot be recreated by applying the reverse rules. Likewise if not all features of the $S_i$ are used by the transformation.
– The forward transformation may not be injective, so that several different source models could produce the same target model.

The reverse rules give a partial language-level interpretation $\chi : L_S \rightarrow L_T$, provided the $Post'_{i,j}$ are explicit and the $SCond_{i,j}$ are exclusive for distinct $j$:

$$\chi((s : S_i | SCond_{i,j})) = \{t : T_{i,j} | TCond_{i,j}\}$$

provided that the $SCond_{i,j}$ partition $S_i$.

– Features $att$ of $S_i$ which are explicitly set to expressions $eatt_{i,j}$ in each $Post'_{i,j}$ are interpreted by

$$if \ self \in T_{i,1} \land TCond_{i,1}[self / t] \ then \ tatt_{i,1}[self / t]$$
$$else \ ...$$
$$else \ if \ self \in T_{i,n} \land TCond_{i,n}[self / t] \ then \ tatt_{i,n}[self / t]$$

where $tatt_{i,j}$ is the $T$ object corresponding to $eatt_{i,j}$, if this is an object, otherwise it is $eatt_{i,j}$.

$\chi$ is therefore partly determined by $\tau$, although different variations on $\chi$ are possible. Essentially $\chi$ reconstructs a model of $S$ from one of $T$, so that if an interpretation exists for each $S_i$ and feature of the source language this shows that $T$ is at least as expressive as $S$, and that $\tau$ is reversible, because all information of the source model has been retained in the target:

– If $\chi(S_i)$ exists for each entity $S_i$ of $S$, then this is an expression of $L_T$ which defines the set of objects of $T_{i,1}, \ldots, T_{i,n}$, etc corresponding to the elements of $S_i$, ie, given an element $t$ we can determine which $S_i$ it interprets.
if $\chi(f)$ exists for each data feature $f$ of each $S_i$ of $S$, then this is an expression built out of terms of $T$, and can be used to explicitly define $f$ in $Post'_{i,j}$ as $t.\chi(f)$ or the $S$ object corresponding to $t.\chi(f)$.

As the following counter-examples show, it is not possible to make further logical connections between the above general properties.

- An example of a reversible transformation which has an interpretation but is not change propagating is given by $S$ consisting of a single entity $Entity$ with one (identity) attribute $name : String$, and the constraint $name.size > 0$. $T$ also has a single entity $Entity1$ with one (identity) attribute $name1 : String$, and an inheritance relationship $general/specific$ between $Entity1$ and itself (both ends are $\ast$). This has the language constraint

$$specific = Entity1 \rightarrow select(e \mid name1 \ll e.name1)$$

(that is, the ‘subclasses’ of $e1$ are the entities whose name has $e1$’s name as a strict suffix ($\ll$)), and the transformation is therefore specified by:

$$\forall e : Entity \cdot \exists e1 : Entity1 \cdot e1.name1 = e.name \text{ and } e1.specific = Entity1 \rightarrow select(e.name \ll name1)$$

The transformation is reversible by abstraction: simply discarding the $specific/generic$ association from the target model produces a model isomorphic to the source. The reverse implication is:

$$\forall e1 : Entity1 \cdot \exists e : Entity \cdot e.name = e1.name1$$

The interpretation $\chi$ is also trivial, mapping $Entity$ to $Entity1$ and $name$ to $name1$. But the transformation is not change-propagating for creation, deletion or feature changes of the source model, since any such change could require examination of all the names of all instances in the target model, to compute the updated inheritance relation there.

- A transformation which is change-propagating but has no interpretation and only a partial inverse, is given by $S$ consisting of a single entity $Entity$ with no attributes, $T$ consisting of a single global integer attribute $result : Integer$, and $Cons$ defined as:

$$result = Entity \rightarrow size()$$

The first example is a refinement transformation, the second is an abstraction.

The form of the $Post$ predicates themselves will often be of one of the following kinds:

- $t.f = e$ where $f$ is some feature of $T_{i,j}$ and $e$ an expression using feature values of $s$.
- $\exists tsb : TSub \cdot Post_{tsb}$ and $t.f = tsb$ to define a subpart of $t$. (Alternatively $\exists tsb : Set(TSub)$ can be used to select a set of subordinate elements, or $\exists tsb : Sequence(TSub)$ to select a sequence).
Conjunctions of implications

\[(\text{Cond}_1 \implies P_1) \text{ and } \ldots \text{ and } (\text{Cond}_r \implies P_r)\]

where the \(P_i\) are of the first two forms.

A special case are equalities \(t.f = T\text{Sub}[s.g{id}]\) which select a single element of \(T\text{Sub}\) or a set of elements, with primary key value(s) \(id\) equal to \(s.g{id}\) (if this is a single value), or in the set \(s.g{id}\) (if it is a collection). The reversed form \(Post'\) is then \(s.g = S\text{Sub}[t.f{id}]\), if source model entity \(S\text{Sub}\) is in 1-1 correspondence with \(T\text{Sub}\) via the identities.

### 3.3 Special cases of conjunctive-implicative form

For each entity \(S_i\) in the source language, there may be several different \(T_{i,j}\) entities on the right-hand side of implications in \(Cons\), even with the same \(S\text{Cond}_{i,j}\) conditions. This may occur because the features of \(S_i\) are being split or distributed between different target language entities, or because complex and separate structures are being generated in the target model from the same source data (akin to the Builder design pattern [7]).

In these cases it is important to define in the separate target entities some key attribute or stereotype which records the fact of the semantic link between them (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 \(att_1, att_2\), and key attribute \(id\), and two target entities each with one of these attributes, but both having the key:

\[
\forall s : S_1 \cdot \exists t : T_1 \cdot t.id = s.id \text{ and } t.att_3 = s.att_1
\]

\[
\forall s : S_1 \cdot \exists t : T_2 \cdot t.id = s.id \text{ and } t.att_4 = s.att_2
\]

The \(\chi\) interpretation of \(S_1\) is \(\{(t_1, t_2) : T_1 \times T_2 \mid t_1.id = t_2.id\}\), and \(att_1\) can therefore be interpreted as \(att_3\) of the first projection on such pairs, and \(att_2\) as \(att_4\) of the second projection on such pairs.

The reverse rules can be simply stated as:

\[
\forall t : T_1 \cdot \exists s : S_1 \cdot s.id = t.id \text{ and } s.att_1 = t.att_3
\]

\[
\forall t : T_2 \cdot \exists s : S_1 \cdot s.id = t.id \text{ and } s.att_2 = t.att_4
\]

as with the general case of the conjunctive-implicative form. Without the identity attributes such definitions would not be possible.

The reverse transformation illustrates another special form, where two source language entities are merged into a single target language entity. In this case both source entities are interpreted by the same target entity.

The tree to graph transformation is another example of the special case, in which \(\{ t : Tree \mid t.parent = t\}\) is interpreted by \(Node = Edge.source\), the set of
nodes which are not the source of any edge, whilst \( \{ t : \text{Tree} \mid t.\text{parent} \neq t \} \) is interpreted by
\[
\{(n, e) : \text{Node} \times \text{Edge} \mid e.\text{source} = n\}
\]

\( \text{parent} \) is interpreted by \( e.\text{target} \) in the second case.

The one \( S_i \) to multiple \( T_j \) case occurs in refinements, such as the generation of multiple J2EE elements (database tables, value objects, EJBs) from single UML classes. A large example carried out by one of the authors on a defence application was the mapping from data-flow diagrams to HOOD [6]. Data stores were mapped to classes, but processes with multiple flows were mapped to multiple operations on the different classes corresponding to the stores incident with the flows.

Another variation on the conjunctive-implicative pattern occurs when the succedent of a constraint \( C_n \) both reads and writes to the same entity or feature:
\[
\text{wr}(C_n) \cap (\text{rd}('Post') \cup \text{rd}(\text{TCond}))
\]
is non-empty, where \( \text{wr}(P) \) is the write frame of a predicate \( P \), and \( \text{rd}(P) \) the read frame (Appendix B). Such constraints are said to be of type 2 (if the antecedent data and \( S_i \) are not in \( \text{wr}(C_n) \)). Type 1 constraints have \( \text{rd}(C_n) \cap \text{wr}(C_n) = \emptyset \).

This means that the order of application of the constraint to instances may be significant, and that a single iteration through the source model elements may be insufficient to establish \( \text{Post} \) for all elements. A least-fixed point computation may be necessary instead, with iterations repeated until no further change takes place. An example is the derivation of the reachability relationship in a state machine (Figure 2). This is a refinement transformation, with the target language extending the source language, by the additional feature \( \text{reaches} \).

The constraint in this case is:
\[
\forall t : \text{Transition} \cdot \{ t.\text{target} \} \cup t.\text{target}.\text{reaches} \subseteq t.\text{source}.\text{reaches}
\]

Here the feature \( \text{reaches} \) is both read and written.

A least fixed point does exist, if \( \text{reaches} \) is initialised to \( \emptyset \), since the \( \text{reaches} \) sets are monotonically increasing, and these sets are each bounded above by the set of all states. Therefore a terminating and confluent solution does exist for this specification. However the form of this solution is different to the simpler case where the sets of data written and read in a constraint are entirely disjoint (type 1 constraints).

A measure \( Q : \mathbb{N} \) on the source and target models is necessary to establish the termination and correctness of the implementation of type 2 constraints, and should be defined together with \( \text{Cons} \). \( Q \) should be decreased by each application of a constraint, and \( Q = 0 \) at termination of the transformation. In this example, \( Q \) can be defined as:
\[
\Sigma_s:\text{State} (\#\text{states reachable from } s) - \#s.\text{reaches}
\]
If for each particular starting state of the source and target models there is a unique possible modified state of the models (produced by applying the constraints) in which \( Q = 0 \), then the implementation is confluent. This is the case in this example.

Likewise, a \( Q \) measure is necessary if \( S_i \in \text{wr}(Cn) \) or \( \text{wr}(Cn) \cap \text{rd}(SCond) \neq \{\} \), a constraint of type 3. An example of such a constraint is the derivation of the transitive closure of a graph [19]:

\[
\forall e_1, e_2 : \text{Edge} \cdot e_1 \neq e_2 \text{ and } e_1.trg = e_2.src \text{ implies } \\
\exists e_3 : \text{Edge} \cdot e_3.src = e_1.src \text{ and } e_3.trg = e_2.trg
\]

For this, \( Q \) can be defined as the number of chains of edges in the original graph without an edge linking the start of the chain to its end.

### 3.4 Recursive forms

The conjunctive-implicative form presents the intended result of the transformation in an explicit form, but without any dependence upon a particular implementation strategy, thus facilitating validation and general comprehension. However in some cases the complexity of a transformation means that such a form is not possible, or it would be too far separated from the practical implementation of the transformation. A recursive form can be used instead, which defines the effect of the transformation as a recursive function (or set of recursive functions), each evaluation step of the function corresponds to the application of some transformation rule on a source model element, the recursion produces a final result when no further rule is applicable.
In effect the specification defines a function

\[ \tau : \text{Models}_S \rightarrow \text{Models}_T \]

directly in terms of the concrete representations of the models as tuples \((S_1, \ldots, f_1, \ldots)\), so that this approach is less abstract than the previous approach. We will assume that the source and target languages \(S\) and \(T\) are the same.

Formally, the \(\text{Cons}\) predicate is defined as

\[ tmodel = \tau(smodel \circ \text{pre}) \]

where \(smodel\) is a tuple comprising the source model, and \(tmodel\) a tuple comprising the target model.

\(\tau\) is defined by a disjunction of constraints:

\[
\begin{align*}
(\exists s_1 : S_1 \cdot S\text{Cond}_1 \text{ and } \exists t_1 : T_{j_1} \cdot \tau(smodel) = \tau(smodel_1)) \text{ or } \\
\ldots \\
(\exists s_n : S_n \cdot S\text{Cond}_n \text{ and } \exists t_n : T_{j_n} \cdot \tau(smodel) = \tau(smodel_n)) \text{ or } \\
(\neg(\exists s_1 : S_1 \cdot S\text{Cond}_1) \text{ and } \ldots \text{ and } (\exists s_n : S_n \cdot S\text{Cond}_n) \text{ and } \\
\tau(smodel) = smodel)
\end{align*}
\]

where the \(smodel_i\) are updated versions of \(smodel\) using the selected domain elements \(s_i\) and generated elements \(t_i\). The left hand conditions \(\exists s_i : S_i \cdot S\text{Cond}_i\) should be pairwise exclusive, to avoid inconsistencies. Alternatively their ordering could be used to define a priority on the rules/constraints.

Quality-improvement transformations often follow this pattern, since they may be schematically specified by disjunctions of clauses

\[
(\exists e_1 : S_1; \ldots; e_n : S_n \cdot S\text{Cond} \text{ and } \exists f_1 : T_i; \ldots; f_m : T_{i_m} \cdot \tau(smodel) = \tau(\text{restr}(smodel)))
\]

where \(S\text{Cond}\) is some condition identifying a lack of the desired quality in a model. For example, the existence of a class with two immediate ancestors, for a transformation that removes multiple inheritance. The restructuring \(\text{restr}\) depends on the \(e_i\) and \(f_j\).

A final clause

\[
(smodel \text{ satisfies all quality conditions and } \tau(smodel) = smodel)
\]

terminates the recursion. Some quality measure \(Q : \mathbb{N}\) should be definable, so that \(Q(\text{restr}(smodel)) < Q(smodel)\) in each clause, and \(Q(smodel) = 0\) in the terminal cases.

\[
Q(smodel) = \sum_{c \text{ root} \in \text{Class}} (c.\text{generalization} \rightarrow \text{size}() - 1)
\]

for the multiple-inheritance removal transformation.
The constraint for the multiple-inheritance removal transformation could be written as follows in a simplified form of this style:

$$\exists c : \text{Class}; \ g : c.\text{generalization} \cdot c.\text{generalization}\rightarrow\text{size()} > 1 \ \text{and} \ g.\text{general}.\text{allFeatures}\rightarrow\text{size()} =$$

$$c.\text{generalization}.\text{general}\rightarrow\text{collect(}c.\text{generalization}\rightarrow\text{size()}\text{)}\rightarrow\text{min()} \ \text{and} \ \exists a : \text{Association} \cdot a.\text{end}1 = c \ \text{and} \ a.\text{end}2 = g.\text{general} \ \text{and} \ a.\text{multiplicity}1 = \text{ONE} \ \text{and} \ a.\text{multiplicity}2 = \text{ZEROONE} \ \text{and} \ g.\text{isDeleted}())$$

meaning that $\tau(\text{Class, Generalization, Association, Property, ownedAttribute, generalization, general, end1, end2, multiplicity1, multiplicity2,...})$ is equal to $\tau$ of the updated model

$$(\text{Class, Generalization} - \{ g \}, \text{Association} \cup \{ a \}, \text{Property, ownedAttribute, generalization} - \{ c \rightarrow g \}, \text{general} - \{ g \rightarrow g.\text{general} \}, \text{end}1 \cup \{ a \rightarrow c \}, \text{end}2 \cup \{ a \rightarrow g.\text{general} \}, \text{multiplicity}1 \cup \{ a \rightarrow \text{ONE} \}, \text{multiplicity}2 \cup \{ a \rightarrow \text{ZEROONE} \}, ...)$$
in the case that the antecedent holds for $c$ and $g$ (ie, that $Q > 0$ and $g$ is some element of minimal semantic weight amongst the generalisations of $c$).

The first representation of the constraint does not make sense as a property which must hold at the end state of a transformation, because $g$ is both assumed to exist in the model and to be removed from the model. Only the formalised version using recursion to define $\tau$ can be interpreted logically. For convenience, however, the informal versions are used in UML-RSDS to define this form of transformation.

It is difficult to establish change-propagation for recursive specifications, because a small change to the source model may affect an arbitrary number of evaluation steps of $\tau$. Likewise, the construction of an interpretation involves consideration of each individual step. The specification may not identify a unique recursive function $\tau$ because the steps may not be confluent, even if the conditions of the cases are disjoint (this is the case for the multiple-inheritance removal transformation). However, inductive reasoning can be used to deduce properties which any solution to the recursion must satisfy, for example, that $\text{Class} = \text{Class}@\text{pre}$ in this transformation, since no step modifies the set of classes.

Other examples of the recursive pattern are the evaluation of OCL expressions using transformations in [22], the mapping of one representation of Lambda-expressions to another [8], slicing of state machines [16], and quality-improvement transformations such as the removal of duplicated attributes in a class diagram, or the replacement of non-abstract superclasses by abstract classes.

3.5 Auxiliary metamodels

An additional specification pattern, which may be used with either the conjunctive-implicative or recursive forms, is the use of an enhanced source or target meta-
model to express the specification. For example, in the state machine slicing transformation of [16], 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. In Figure 2 we have drawn the auxiliary metamodel element using dashed lines. Likewise for the lambda-calculus restructuring transformation [8], an additional association recording the set of bound variables in scope in each expression is required. Auxiliary metamodels are also used to implement tracing facilities, in transformation languages such as VIATRA [27] and Kermeta [10], 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.

4 Implementation Patterns

Implementation of a model transformation may be carried out by the use of a special-purpose model transformation language such as ATL [9] or Kermeta [10], or by production of code in a general purpose language such as Java [15]. In either case, the implementation needs to be organised and structured in a modular manner, and ideally it should be possible to directly relate the implementation to the specification, expressed in the Cons predicate.

We will use a small procedural language including assignment, conditionals, operation calls and loops to allow platform-independent imperative definitions of behaviour for transformation implementations (Figure 3). This language corresponds to a subset of UML structured activities.

Fig. 3. Statement metamodel
4.1 Implementation strategies

We will consider three alternative strategies for implementation of a model transformation specification defined by a conjunctive-implicative or recursive pattern:

- **Change-propagation**: the constraints are used directly to implement the transformation, by interpreting them in an operational form. If a change is made to the source model, any constraint whose lhs is made true by the change is applied, and the effects defined by its rhs are executed to modify the target model.

- **Recursive**: the transformation is initiated by construction of instances of the topmost entities in the entity hierarchy of the target language $T$, such construction may require recursive construction of their components, etc.

- **Layered**: the base elements (instances of the entities lowest in the entity hierarchy of $T$) are constructed first, then elements that depend upon these are constructed, etc.

It is possible to perform the layering from the top down, as in the Viatra version of the UML to relational database mapping [27], although this means that target model elements may be incomplete and invalid during the intermediate steps.

Both the recursive and layered approaches make use of a partial order on the set of target language entities $T_l$ [29], with instances of entities higher in the order being composed of (or being dependent upon) instances of entities below them in the order. In terms of the conjunctive-implicative form, this means that the constraints creating instances of $T_k$ refer to an entity $T_l$ (or a sub-or super-class of $T_l$) in the $TCond$ or $Post$ predicates to define the $T_k$ instances: $T_k$ is then above $T_l$ in the ordering. Formally, $T_l < T_k$ if $T_k \in wr(Cn)$ and $T_l \in rd(Cn)$ for some constraint $Cn$.

We will consider each strategy in detail in the following sections.

4.2 Change-propagation implementation

For transformations specified using the conjunctive-implicative form, it is in principle possible to interpret the constraints in an operational manner and to directly implement them using some constraint-driven development tool such as UML-RSDS [15], provided that the conditions for change-propagation hold:

- The expressions $SCond$, $TCond$ and $Post$ are localised to $s$, $t$ and $s$, respectively, can be effectively computed, and $Post$ is explicit.

  Lookup of objects by primary key value: $T_k[e]$ can be considered localised and effective if $e$ is.

- There are 1-1 isomorphisms between source entity sets $S_i$ and corresponding target entity sets $\bigcup_j \{ t : T_{i,j} \ | \ TCond_{i,j} \}$, induced by equal identity attribute values.
These conditions enable addition and attribute-change modifications to the source to be propagated to the target. The existential quantifier $\exists t : T_{i,j}$ on the right hand side of a constraint can be interpreted as “if there does not exist an element $t : T_{i,j}$ in the target model satisfying the conditions, create it and set its attributes using $T\text{Cond}_{i,j}$ and $Post_{i,j}$” if $T_{i,j}$ is concrete. This interpretation is also used for $\exists t : T_{i,j}$ if $T_{i,j}$ has an identity attribute – creation of two $T_{i,j}$ objects with the same value for this attribute would violate the constraints of $T$. We call this the unique instantiation pattern. QVT has a similar mechanism, check-before-enforce, to reuse target model elements that satisfy a relation, in preference to recreating new elements [23]. Otherwise the $\exists t : T_{i,j}$ quantifier does not check before creating a $t$ instance for concrete $T_{i,j}$.

4.3 Recursive implementation

This strategy is typically used in transformation languages such as QVT and ATL, via explicit or implicit invocation of relations from other relations (the where clause in QVT). For example, the QVT version of the UML to relational database mapping initiates processing by mapping packages to schemas, then maps the classes of individual packages to relational tables, in turn this involves the mapping of attributes to columns, and finally of UML types to database types [24]. Such a strategy requires that a strict hierarchy exists between the target language entities, i.e., there should be a partial order $E_1 < E_2$ meaning that $E_2$ is composed of $E_1$ elements (directly or indirectly).

In this example,

$Column < Table < Schema$

At each stage the superordinate target entity instances are passed down to the rules defining the subordinate entity instances, for example in our pseudocode notation we could express this transformation as follows:

$$mapPackageToSchema(p : Package) : Schema (result.name := p.name; for c : p.classes do mapClassToTable(c, result); return result)$$

$$mapClassToTable(c : Class, s : Schema) (t : Table; t.name := c.name; t.schema := s; for a : c.allAttribute do mapAttributeToColumn(a, t))$$

$$mapAttributeToColumn(a : Attribute, t : Table) (cl : Column; cl.name := a.name; cl.table := t; cl.type := typeConvert(a.type))$$
This approach could be termed the *recursive descent* pattern. This is effective if the object hierarchy in $T$ does not involve sharing of objects, so that the recursive descent of the corresponding $S$ hierarchy can create $T$ structures depending only on subordinate and superordinate objects in the object hierarchy. Here, although attributes may be in common between some source classes, distinct copies of the corresponding columns are required in the target model. If instead, target elements need to be shared, some lookup mechanism is required to match source model elements and target model elements generated from them (this lookup is performed implicitly in ATL).

Where an entity depends upon itself (as for trees in the tree to graph example), recursion down the object structure (using the inverse relation to *parent* in this case) can be used if this structure is hierarchical.

A technique for synthesising a recursive implementation from a declarative specification using constructive proofs based on partial orderings of metamodel entities is given in [29].

### 4.4 Layered implementation

The layered approach has the advantage that it can be carried out in a *phased* manner: by the sequential composition of separate sub-transformations, each responsible for adding one more layer of structure to the target model, assuming that previous phases have already constructed the lower layers. This implementation of the layered approach could be called the *phased creation* pattern. This strategy reduces the complexity of the transformation compared to the recursive approach, and increases the modularity, by reducing calling dependencies, and makes verification more simple, in particular, proof of termination. However it may be inefficient, because it requires the lookup of constructed elements of a lower level, each time a new element of a higher level is created. Such a lookup may typically be done by a key-based search, or by using an explicit transformation trace facility, as in Kermeta [10] and VIATRA [27].

The UML-RSDS tools efficiently implement searches for the set of elements $T_{Sub}[sexp.id1]$ of a target type $T_{Sub}$ with primary key values in $sexp.id1$ by maintaining a map from the primary keys of $T_{Sub}$ objects to the objects – this could be termed the *object indexing* pattern.

In some cases no clear hierarchy of target language entities exists, for example, there may be mutually dependent entities $T_1$ and $T_2$ in the target model. In such a case a phased approach can first construct all the $T_1$ and $T_2$ instances, in any order, then establish the links (in both directions) between them. For example, if $T_1$ has a feature $att : T_2$ and $T_2$ has a feature $role : Set(T_1)$ and these are not mutual inverses.

This approach can also be used if a target entity $T_j$ depends on itself, or we can use some hierarchical structure for the set of $T_j$ instances in order to implement layered construction of $T_j$ objects. Phased implementation can be used to separate a phase of object construction from a ‘cleanup’ phase where unwanted objects are deleted, the *construction and cleanup* pattern.
A hybrid approach is possible, whereby the layered approach is primarily used, but recursive construction of exclusively-owned subparts $T_{Sub}$ of an entity $T$ is used. Such subparts may typically be entities which do not have a separate identity attribute of their own, so lookup by identity cannot be used for them.

For a phased implementation, the form of the specification can be used to define the individual transformation rules. If $Cons$ is a conjunction of constraints $C_i$ of the form

$$\forall s : S_i \cdot SCond \ implies \ \exists t : T_j \cdot TCond \ and \ Post$$

then each constraint may be mapped individually into a phase $P_i$ which implements it, provided that there are no circularities in the data dependency relationship between constraints. We define $C_i < C_j$ if

$$wr(C_i) \cap rd(C_j) \neq \{\}$$

This should be a partial order, and should imply that $i < j$.

The phases can be executed in any order that is consistent with this data dependency ordering $C_i < C_j$, including concurrent execution if the phases are not related by data dependency. $P_j$ will establish $C_j$ at its termination, under the assumption that $C_i$ holds for each $C_i < C_j$, ie, that each corresponding $P_i$ has been completed and that $C_i$ has not been invalidated by succeeding phases. This can be ensured if the write frames of distinct constraints are disjoint.

If a group of constraints are mutually data dependent, then they must be implemented by a single phase.

In the simple case where a constraint satisfies the non-interference condition:

$$wr(C_i) \cap rd(C_i) = \{\}$$

the (type 1) constraint can be implemented by a loop

$$\text{for } s : S_i \text{ do } s.op_i()$$

where in $S_i$ we include an operation of the form:

$$op_i()$$

$$\text{post: } SCond[\text{self}/s] \ implies \ \exists t : T_j \cdot TCond \ and \ Post[\text{self}/s]$$

We refer to this strategy as constraint implementation approach 1. The proof of correctness of this strategy uses the property that the inference rule: from $v:s \Rightarrow [\text{acts}(v)]P(v)$ derive

$$[\text{for } v : s \text{ do } \text{acts}(v)][\forall v : s@prev \cdot P(v)]$$

is valid for such iterations, provided that one execution of $\text{acts}$ does not affect another: the precondition of each $\text{acts}(v)$ has the same value at the start of
acts(v) as at the start of the loop, and if acts(v) establishes P(v) at its termination, P(v) remains true at the end of the loop [15]. Confluence also follows if the updates of written data in different executions of the loop body are independent of the order of the executions.

For the tree to graph example, C1 < C2, so the phase creating nodes must precede that creating edges. Phase 1 uses the operation

\[\text{mapTreeToNode()}\]

post:
\[\exists \ n : \ Node \ \cdot \ n.name = \text{name}\]

of Tree to map a tree to a node. Since the updates to \(\overline{\text{Node}}\) and \(\text{Node }:: \text{name}\) are independent for distinct \(n\), the implementation will be confluent.

The activity for this phase is a simple unordered iteration over all tree elements:

\[\text{for } t : \text{Tree } \text{do } t.\text{mapTreeToNode()}\]

This iteration executes \(\text{mapTreeToNode()}\) exactly once for each \(t\) in \(\text{Tree}\) at the start of the loop execution. It can also be denoted by \(\text{Tree.mapTreeToNode()}\).

Phase 2 creates edges, assuming that phase 1 has been completed. For phase 2 the operation is:

\[\text{mapTreeToEdge()}\]

post:
\[\forall \ s : \ S_i \cdot \text{SCond implies } \exists \ t : \ T_j \cdot \text{TCond and Post}\]

where this is the only \(S_i\) constraint.

In the case where
\[\text{wr}(C_i) \cap (\text{rd}(\text{Post}) \cup \text{rd}(\text{TCond}))\]
is non-empty but the other conditions of non-interference still hold (a type 2 constraint), an iteration of the form:

\[\text{running} := \text{true};\]

while (running) do
\[\text{(running} := \text{false};\]
\[\text{for } s : \ S_i \text{ do}\]
\[\text{if } \text{SCond} \text{ then}\]
\[\text{if } \text{Succ} \text{ then skip}\]
\[\text{else (s.op(); running} := \text{true})\]
\[\text{else skip}\]

A more complex implementation strategy is required if the non-interference condition does not hold. Consider a constraint \(C_i\):

\[\forall s : \ S_i \cdot \text{SCond implies } \exists t : \ T_j \cdot \text{TCond and Post}\]

This is also iterated over \(\text{Tree}\), and is also confluent.
can be used, where $Succ$ is $\exists t : Tj \cdot TCond$ and $Post$ and $op()$ is defined as:

$$op()$$

$$post: Succ[self/s]$$

(We refer to this strategy as constraint implementation approach number 2). The conditional test of $Succ$ can be omitted if it is known that $SCond \Rightarrow not(Succ)$.

If the updates to the written data are monotone and bounded (e.g., as in the case of the inclusion operator $\subseteq$ for the reachability computation example), then the iteration terminates and computes the least fixed point. A measure $Q : \mathbb{N}$ can be used to prove termination, a suitable measure in this case is the sum $\Sigma_{s,State}(\#states \text{ reachable from } s) - \#s.reaches$. $Q$ is decreased by each application of $op$ and is never increased.

$Q$ is also necessary to prove correctness: while there remain $s : S_i$ with $SCond$ true but $Succ$ false, then $Q > 0$ and the iteration will apply $op$ to such an $s$. At termination the constraint therefore holds true, and $Q = 0$. Confluence also follows if $Q = 0$ is only possible in one unique state of the source and target models which can be reached from the initial state by applying the constraint.

For the reachability computation, the inner loop iterates:

```plaintext
if \{ t.target \} \cup t.target.reaches \subseteq source.reaches
then skip
else (t.op(); running := true)
```

If the other conditions of non-interference fail (a type 3 constraint), then the application of a constraint to one element may change the elements to which the constraint may subsequently be applied to, so that a fixed for-loop iteration over these elements cannot be used. Instead, a schematic iteration of the form:

```plaintext
while some source element s satisfies a constraint lhs do
    select such an s and apply the constraint
```

can be used. This can be explicitly coded as:

```plaintext
running := true;
while running do
    running := search()
```

where:

```plaintext
search() : Boolean
    (for s : S_i do
        if SCond then
            if Succ then skip
            else (s.op(); return true));
    return false
```
and where \( op \) applies the constraint succedent. We call this approach 3, iteration of a search-and-return loop. The conditional test of \( Succ \) can be omitted if it is known that \( SCond \Rightarrow \neg(Succ) \). As in approach 2, a \( Q \) measure is needed to prove termination and correctness. \( search \) returns false exactly when \( Q = 0 \). Again, confluence can be deduced from uniqueness of the termination state.

For the multiple-inheritance removal transformation, the search operation is:

\[
\text{(for } c : \text{ Class } \text{ do}
\begin{align*}
\text{for } g : \text{ c.generalization do} & \\
\text{if } c\text{.generalization}\rightarrow\text{size()} & > 1 \text{ and} \\
\text{g.general.allFeatures}\rightarrow\text{size()} & = \\
\text{c.generalization.general}\rightarrow\text{collect(allFeatures}\rightarrow\text{size()}\rightarrow\text{min()} \\
\text{then } (c\text{.op(g); return true}); \\
\text{return false}
\end{align*}
\]

The operation is applied repeatedly to remove multiple inheritances from the model, until no case remains (ie, when \( Q = 0 \)). Here the termination states are not unique and confluence fails.

For quality-improvement transformations the \( Q \) metric can be used to select constraints: the constraint application which maximally reduces this measure can be chosen if several are possible.

In summary, we have identified the following implementation patterns:

- **Unique instantiation**: before creating an instance to satisfy \( \exists_1 t : TType \cdot TPred \), search to see if one already exists, and use this if so. This is related to the Singleton pattern [7] and may use Object indexing.
- **Object indexing**: to find collections \( TType[ids] \) of \( TType \) elements with a given set \( ids \) of primary key values, maintain a map from primary keys to \( TType \) elements.
- **Recursive descent**: recursively map subcomponents of a source model element as part of the mapping operation of the element, passing down the target object(s) in order to link subordinate target elements to it/them.
- **Phased creation**: before creating an instance of an entity \( T_1 \), create all instances of entities \( T_2 \) hierarchically below \( T_1 \), which are mapped to by the transformation. In the case of mutually dependent entities, create all objects of the entities before setting the links between them.
- **Construction and cleanup**: separate construction of new elements from the removal of deleted elements, placing these processes in successive phases.

The Builder and Abstract Factory patterns are also directly relevant to pattern implementation, in cases where complex platform-specific structures of elements must be constructed from semantic information in a platform-independent model, such as the synthesis of J2EE systems from UML specifications.

5 Formalisation of model transformation design

The process of deriving designs for a model transformation from its specification can be formalised and expressed itself as a (meta) model transformation at the
M2 level. The key parts of the source and target languages are given in Figure 4, the statement language is defined in Figure 3. Transformations are considered to be particular use cases of a system. Use cases may have pre and postconditions (Ens and Cons constraint sets for transformations) and invariant properties (Pres for transformations).

We have extended the UML metamodel with auxiliary associations orderedPostconditions and implementedBy, and auxiliary query operations to return the read and write frame of constraints, and to return the main quantified class and variable, secondary quantified variables, and the quantified body of the constraint.

Individual constraints $C_n$:

$$\forall s : S_i \cdot SCond \implies \exists t : T_j \cdot TCond \& Post$$

are examined to identify which implementation strategy can be used to derive their design. This depends upon the features and objects read and written within the constraint (Table 1).

The set of constraints are also examined to determine the dependency relation $<$ between constraints. It is assumed that the constraints are consistent, and that the specifier has ordered the constraints so that $C_1$ precedes $C_2$ if $rd(C_2) \cap wr(C_1)$ is non-empty. If a group of constraints are mutually data-dependent, then they are mapped into a single phase, implemented using approach 2 or 3.

In the case that there are two mutually dependent constraints with outer quantifier $\forall s : S_i$, approach 2 has the form

$$running := true;$$
$$while (running) do$$
$$\quad (running := false;$$
$$\quad \quad for s : S_i do loop)$$

where $loop$ is the code:

$$if SCond1 then$$
<table>
<thead>
<tr>
<th>Type 1 constraint</th>
<th>Constraint properties</th>
<th>Implementation choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>No interference between different applications of constraint, and no change to $S_i$ or $rd(SCond)$:</td>
<td>Approach 1: single for loop</td>
<td></td>
</tr>
<tr>
<td>$wr(Cn) \cap rd(Cn) = {}$</td>
<td>$for\ s:S_i\ do\ s.op()$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type 2 constraint</th>
<th>Constraint properties</th>
<th>Implementation choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interference between different applications of constraint, but no update of $S_i$ or $rd(SCond)$ within constraint: $S_i \notin wr(Cn)$, $wr(Cn) \cap rd(SCond) = {}$</td>
<td>Approach 2: while iteration of for loop</td>
<td></td>
</tr>
<tr>
<td>$Q$ measure needed for termination and correctness proof</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type 3 constraint</th>
<th>Constraint properties</th>
<th>Implementation choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update of $S_i$ or $rd(SCond)$ within constraint. $S_i \in wr(Cn)$, or $wr(Cn) \cap rd(SCond) \neq {}$</td>
<td>Approach 3: while iteration of search-and-return for loop</td>
<td></td>
</tr>
<tr>
<td>$Q$ measure needed for termination and correctness proof</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Design choices for constraints

```plaintext
if Succ1 then skip
else (s.op1(); running := true);
if SCond2 then
  if Succ2 then skip
  else (s.op2(); running := true)
```

$op1$ implements the succedent of the first constraint, and $op2$ that of the second. Similar extensions can be made for approach 3, and for approach 2 and 3 with distinct source entities for the outer quantifier.

The generated code of each phase can then be combined in an overall sequence statement.

This meta-transformation itself follows the conjunctive implicative pattern. For example, the constraint defining the construction of simple iterations for type 1 constraints $\forall\ s:S_i \cdot SCond \implies \exists\ t:T_j \cdot TCond$ and $Post$ is:

$$\forall\ u:UseCase;\ c:u.orderedPostconditions \cdot \begin{align*}
  \&\ & rd(c) \cap wr(c) = \{} & \text{and} \quad secondaryVars(c) = Sequence\{} & \implies \\
  \&\ & \exists\ d:Operation \cdot d.name = u.name + c.name & \text{and} \\
  \&\ & \qquad d:primaryClass(c).ownedOperation & \\
  \&\ & \qquad body(c).substitute("self", primaryVar(c)) : d.postcondition & \\
  \&\ & \exists\ stat:OperationCallStatement \cdot \\
  \&\ & \qquad stat.calledOperation = d & \\
  \&\ & \qquad stat.target = primaryClass(c).allInstancesExp() & \\
  \&\ & \qquad c.implementedBy = stat
\end{align*}$$

$cl.allInstancesExp()$ returns the OCL $CallExp$ representing $S_i.allInstances()$, if $cl$ represents $S_i$. 

In a final phase, all constraint implementations are gathered into a sequence statement:

\[ \forall u : \text{UseCase} \cdot \exists \text{stat} : \text{SequenceStatement} \cdot \\
\text{stat.statements} = u.\text{orderedPostconditions.implementedBy} \quad \text{and} \\
\text{stat} : u.\text{classifierBehavior} \]

Use cases are a subclass of \text{BehavioredClassifier} in UML, so they have a \text{classifierBehavior}. We use this to define the overall activity which implements the use case.

The transformation is an update-in-place transformation, in which all the constraints are of the first kind in Table 1. In the UML-RSDS tools it is implemented in a change-propagating manner: when a new constraint is added to a use case, the implementing operations and activities of the use case are automatically updated. We intend to also support the generation of checking transformations from use case preconditions (\text{Asm} constraints) and inverse transformations from the \text{Cons} constraints of invertible transformations.

The UML-RSDS tools, together with examples of transformation specification and implementation, are available at: \url{http://www.dcs.kcl.ac.uk/staff/kcl/uml2web/}.

6 Related Work

In [3], specifications of the conjunctive-implicative form are derived from model transformation implementations in triple graph grammars and QVT, in order to analyse properties of the transformations, such as definedness and determinacy. This form of specification is therefore implicitly present in QVT and other transformation languages, and we consider that it is preferable for such logical specifications to be defined prior to detailed coding of the transformation rules, in order to identify possible errors at an earlier development stage.

In [28] the concept of the conjunctive-implicative form was introduced to support the automated derivation of transformation implementations from specifications written in a constructive type theory. A single undecomposed constraint of the \( \forall \Rightarrow \exists \) form defines the entire transformation, and the constructive proof of this formula produces a function which implements the transformation. In [29] the conjunctive-implicative form is used as the basis of model transformation specifications in constructive type theory, with the \( \exists x. P \) quantifier interpreted as an obligation to construct a witness element \( x \) satisfying \( P \). A partial ordering of entities is used to successively construct such witnesses. In this paper we show how this approach may be carried out in the context of first order logic, with systematic strategies and patterns used to derive transformation implementations from transformation specifications, based on the detailed structure of the specifications. The correctness of the strategies and patterns are already established and do not need to be re-proved for each transformation, although side conditions (such as the data-dependency conditions and existence of a suitable \( Q \) measure) need to be established.
In [1], a transformation specification pattern is introduced, *Transformation parameters*, to represent the case where some auxiliary information is needed to configure a transformation. This could be considered as a special case of the auxiliary metamodel pattern. An implementation pattern *Multiple matching* is also defined, to simulate rules with multiple element matching on their antecedent side, using single element matching. We also use this pattern, via the use of multiple $\forall$ quantifiers in specifications and multiple *for* loops at the design level to select groups of elements. Our work extends previous work on model transformation patterns by combining patterns into an overall process for developing model transformation designs and implementations from their specifications.

7 Summary

We have described three specification patterns: the conjunctive-implicative and recursive forms of specification constraints and the auxiliary metamodel pattern, and five implementation patterns. We have described how the form of specification constraints can be used to deduce global properties of a transformation, and can be used to construct an implementation. Quality metrics could also be based on the patterns, for example measures of the degree of locality of data accesses in the specification of a model transformation would be relevant to the efficiency and modularity of the transformation.

Acknowledgement

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

References

7. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
A Formal statement of patterns

In this section we define the transformation patterns using the standard GoF pattern documentation format [7].

A.1 Conjunctive implicative form

Synopsis To specify the effect of a transformation in a declarative manner, as a global pre/post predicate, consisting of a conjunction of constraints with a $\forall \Rightarrow \exists$ structure.
**Forces** Useful whenever a platform-independent specification of a transformation is suitable. The conjunctive-implicative form can be used to analyse the semantics of a transformation, and also to construct an implementation.

**Solution** The Cons predicate should be split into separate conjuncts each relating one (or a group) of source model elements to one (or a group) of target model elements:

$$
\forall s : S_i \cdot SCond_{i,j} \implies \exists t : T_i, j \cdot TCond_{i,j} \text{ and Post}_{i,j}
$$

where the $S_i$ are source entities, the $T_{i,j}$ are target model entities, $SCond_{i,j}$ is a predicate on $s$ (identifying which elements the constraint should apply to), and $Post_{i,j}$ defines $t$ in terms of $s$.

For Cons specifications including type 2 or type 3 constraints, suitable $Q$ measures are needed to establish termination, confluence and correctness of the derived transformation implementation.

**Consequences** The Ens and Pres properties should be provable directly from the constraints: typically by using the Cons constraints that relate the particular entities used in specific Ens or Pres constraints.

**Implementation** Either by the change-propagation, layered or recursive implementation strategies.

**Code examples** A large example of this approach for a migration transformation is in [17]. The UML to relational mapping is also specified in this style in [18]. In this paper we have described the tree to graph transformation in detail. In this case Ens1 follows from $C1$, $C2$ and the uniqueness of name on trees and nodes, Ens2 follows from $C2$.

### A.2 Recursive form

**Synopsis** To specify the effect of a transformation in a declarative manner, as a global pre/post predicate, using a recursive definition of the transformation relation.

**Forces** Useful whenever a platform-independent specification of a transformation is required, and the conjunctive-implicative form is not applicable, because the post-state of the transformation cannot be directly characterised or simply related to the pre-state.

**Solution** The Cons predicate is defined by an equation such as

$$
tmodel = \tau(smodel@pre)
$$

or more generally as

$$(smodel, tmodel) = \tau(smodel@pre, tmodel@pre)
$$

where $smodel$ represents the source model and $tmodel$ the target model, and $\tau$ is a recursive function defined by a disjunction of clauses

$$
\exists s : S_i \cdot SCond_{i,j} \text{ and } \exists t : T_{i,j} \cdot TCond_{i,j} \text{ and Post}_{i,j}
$$
where the \( S_i \) are source entities, the \( T_{i,j} \) are target model entities, \( SCond_{i,j} \) is a predicate on \( s \) (identifying which elements the constraint should apply to), and \( Post_{i,j} \) defines the mapping \( \tau(smodel, tmodel) \) in terms of \( smodel, tmodel, s, t \) and other mapping forms \( \tau(smodel', tmodel') \) for some modified models \( smodel', tmodel' \) derived from \( smodel, tmodel \). The \( \exists s : S_i \cdot SCond_{i,j} \) conditions should be pairwise exclusive, or the ordering of the constraints used to define a priority ordering of tests.

There should exist a measure \( Q : \mathbb{N} \) on the state of a model, such that \( Q \) is decreased on each step of the recursion, and with \( Q = 0 \) being the termination condition of the recursion. A default case with conclusion \( \tau(smodel, tmodel) = (smodel, tmodel) \) applies in this case. \( Q \) is an abstract measure of the time complexity of the transformation, the maximum number of steps needed to complete the transformation on a particular model. For quality-improvement transformations it can also be regarded as a measure of the (lack of) quality of a model.

**Consequences** The proof of \( Ens \) and \( Pres \) properties from \( Cons \) is more indirect for this style of specification, typically requiring induction using the recursive definitions.

**Implementation** The constraints can be used to define a recursive operation that satisfies the specification, or an equivalent iterative form. The constraints can also be used to define pattern-matching rules in transformation languages such as ATL or QVT.

**Code examples** Many computer science problems can be expressed in this form, such as sorting, searching and scheduling. For example, the sorting of a list \( s \) of integers ordered by \( < \) is:

\[
(\exists i, j : 1..s.size() \cdot i < j \text{ and } s(j) < s(i) \text{ and } \tau(s) = \tau(s \oplus \{ i \mapsto s(j), j \mapsto s(i) \}) \text{ or } \\
((\forall i, j : 1..s.size() \cdot i < j \implies s(i) \leq s(j)) \text{ and } \tau(s) = s)
\]

The quality measure \( Q(s) \) is the number of disorderings in \( s \), i.e., the number of pairs \( i, j \) of indexes satisfying the antecedent of the first constraint. The recursion terminates when this reaches 0:

\[
Q(s) = 0 \implies \tau(s) = s
\]

\( Q(s) \) in the worst case is of order \( n^2 \), where \( n \) is the size of \( s \).

An iterative implementation of this problem in Java could be:

```java
boolean running = true;
while (running)
{ running = rule1(s); }
```

where:

```java
public static boolean rule1(int[] s)
{ for (int i = 0; i < s.size(); i++)
{ for (int j = i+1; j < s.size(); j++)
{ if (s[j] < s[i])
{ int temp = s[i];
 s[i] = s[j];
 ```
s[j] = temp;
return true;
}
}
return false;

We use the implementation approach 3, as for type 3 constraints, to implement recursive specifications in Java.

A.3 Auxiliary metamodel

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

Forces Useful whenever auxiliary data needs to be used in a transformation: such data may simplify the transformation definition, and may permit a more convenient use of the transformation, eg., by supporting decomposition into sub-transformations. A typical case is a query transformation which counts the number of instances of a complex structure in the source model: explicitly representing these instances as instances of a new (auxiliary) entity may simplify the transformation.

Tracing can also be carried out by using auxiliary data to record the history of transformation steps within a transformation.

Solution Define the auxiliary metamodel as a set of (meta) attributes, associations, entities and generalisations extending the source and/or target metamodels. These elements may be used in the succedents of Cons constraints (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).

Consequences 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).

Code example An example is a transformation which returns the number of cycles of three distinct nodes in a graph. This problem can be elegantly solved by extending the basic graph metamodel by defining an auxiliary entity ThreeCycle which records the 3-cycles in the graph (Figure 5). The auxiliary language elements are shown with dashed lines.

The specification Cons of this transformation then defines how unique elements of ThreeCycle are derived from the graph, and returns the cardinality of this type at the end state of the transformation:

(C1):

\( \forall g : \text{Graph} \cdot \forall e1 : g.\text{edges}; e2 : g.\text{edges}; e3 : g.\text{edges} \cdot \\
\quad e1.\text{trg} = e2.\src \text{ and } e2.\text{trg} = e3.\src \text{ and } e3.\text{trg} = e1.\src \text{ and } \\
\quad (e1.\src \cup e2.\src \cup e3.\src) \rightarrow \text{size}() = 3 \implies \\
\exists tc : \text{ThreeCycle} \cdot tc.\text{elements} = (e1.\src \cup e2.\src \cup e3.\src) \text{ and } \\
\quad tc : g.\text{cycles} \)
Fig. 5. Extended graph metamodel

(C2) : ∀ g : Graph · ∃ r : IntResult · r.num = g.cycles→size()

The alternative to introducing the intermediate entity would be a more complex definition of the constraints, involving the construction of sets of sets using OCL collect.

Related patterns This pattern extends the conjunctive-implicative and recursive form patterns, by allowing constraints to refer to data which is neither part of the source or target languages.

A.4 Construction and cleanup

Synopsis To simplify a transformation by separating rules which construct model elements from those which delete elements.

Forces Useful when a transformation both creates and deletes elements of entities, resulting in complex specifications.

Solution Separate the creation phase and deletion phase into separate constraints, 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.

Consequences 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.

Code examples An example is migration transformations where there are common entities between the source and target languages [20]. 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.
Another example are complex update-in-place transformations, such as the removal of duplicated attributes [21]. A rule (in the recursive style) which moves attributes duplicated in all direct subclasses of a class up to the class, could be:

\[
\forall c : \text{Class} \mid c.\text{specialization}\rightarrow\text{size()} > 1 \implies
\forall a : c.\text{specialization}\_\text{specific}.\text{ownedAttribute} :
\]

\[
c.\text{specialization}\_\text{specific}\rightarrow\text{forAll}(
\text{ownedAttribute}\rightarrow\exists(x = a.\text{name} \land \text{type} = a.\text{type}) \implies
\exists p : \text{Property} : p.\text{name} = a.\text{name} \land p.\text{type} = a.\text{type} \land
p : c.\text{ownedAttribute} \land
\text{c.specialization}\_\text{specific}\rightarrow\text{forAll}(a.\text{name} \notin \text{ownedAttribute}.\text{name})
\]

This both creates a new attribute of the superclass with the duplicated name and type, and removes the attribute copies from the subclasses. Instead, the constraint can be rewritten to only create \(p\), and a separate constraint defined to subsequently remove the copies:

\[
\forall c : \text{Class}; p : c.\text{allAttribute}; c_1 : c.\text{specialization}\_\text{specific} : 
\]

\[
c.\text{ownedAttribute}\rightarrow\text{select}(x = p.\text{name})\rightarrow\text{isDeleted}()
\]

\(e\rightarrow\exists(x \mid P)\) is OCL syntax for \(\exists x : e \mid P, e\rightarrow\text{forAll}(x \mid P)\) is OCL syntax for \(\forall x : e \cdot P\).

Another implementation strategy for this pattern is to explicitly mark the unwanted elements for deletion in the first phase, and then to carry out the deletion of marked elements in the second phase. In this example an additional flag attribute of \(\text{Property}\) could be used to indicate which subclass properties should be removed. A similar approach can be applied to the transformation to remove multiple inheritance: a first phase could mark the generalisations to be removed, a second phase could introduce the replacement associations, and a third deletes the marked generalisations.

A.5 Unique instantiation

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

Forces Required when duplicated copies of objects in the target model are forbidden, either explicitly by use of the \(\exists t : T_j \cdot \text{Post}\) quantifier, or implicitly by the fact that \(T_j\) possesses an identifier (primary key) attribute.

Solution To implement a specification \(\exists t : T_j \cdot \text{Post}\) for a concrete class \(T_j\), test if \(\exists t : T_j \cdot \text{Post}\) is already true. If so, take no action, otherwise, create a new instance \(t\) of \(T_j\) and establish \(\text{Post}\) for this \(t\).

In the case of a specification \(\exists t : T_j \cdot t.\text{id} = x \land \text{Post}\) where \(\text{id}\) is a primary key attribute, check if a \(T_j\) object with this \(\text{id}\) value already exists: \(x \in T_j.\text{id}\) and if so, use the object \((T_j[x])\) to establish \(\text{Post}\).

Consequences The pattern ensures the correct implementation of the constraint. 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.
Implementation The executable ‘update form’ in Java of $\exists t : T_j \cdot Post$ for a concrete class $T_j$ is:

```java
if (qf) { }
else
{ uf }
```

where $qf$ is the query form of $\exists t : T_j \cdot Post$, and $uf$ is its update form.

The update form of $\exists t : T_j \cdot t.id = x$ and $Post$ for a class $T_j$ with primary key $id$ is:

```java
Tj t;
if (qf)
{ t = Controller.inst().getTjByPK(x'); }
else
{ uf; }
Post*
```

where $qf$ is the query form of $x : T_j.id$, $x'$ the query form of $x$, $Post*$ the update form of $Post$, and $uf$ creates a new instance $t$ of $T_j$ with $t.id = x$.

Code example Here is sample code, created to check the uniqueness of a ThreeCycle in a graph, corresponding to the succedent of constraint $C1$ above:

```java
if (Set.exists_0(Controller.inst().threecycles, this, e1, e2, e3)) {} else
{ ThreeCycle tc = new ThreeCycle();
 Controller.inst().addThreeCycle(tc);
 Controller.inst().addcycles(this,tc);
 Controller.inst().setelements(tc,Set.union(e1.getsrc(),
 Set.union(e2.getsrc(),e3.getsrc())));
}
```

where the $exists_0$ query operation tests if there is already a three cycle with the given elements in $this.cycles$.

The ‘check before enforce’ matching semantics for QVT rules is another example of the pattern.

Related patterns Object indexing can be used to efficiently obtain an object with a given primary key value in the second variant of the pattern.

A.6 Object indexing

Synopsis All objects of a class are indexed by a unique key value, to permit efficient lookup of objects by their key.

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

Solution Maintain an index map data structure $cmap$ of type $IndType \rightarrow C$, where $C$ is the class 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 $v$: $cmap.get(v)$.
Consequences 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.

Implementation When a new C object c is created, add c.ind → c to cmap. When c is deleted, remove this pair from cmap. To look up C objects by their id, apply cmap.

An alternative strategy to look up target model elements is to use an explicit transformation trace facility, as in Kermeta [10] and VIATRA [27].

Code example Here is sample code, as generated by the UML-RSDS tools for an entity Person with primary key pid : String.

In the Controller class representing the system, there are attributes

List persons = new Vector(); // of Person
Map personindex = new HashMap(); // String -> Person

representing the set of existing Person objects, and the indexing map. This class has operations:

public void addPerson(Person p)
{ persons.add(p);
  personindex.put(p.getpid(),p);
}

public void removePerson(Person p)
{ persons.remove(p);
  personindex.remove(p.getpid());
}

public Person getPersonByPK(String pind)
{ return (Person) personindex.get(pind); }

public List getAllPersonByPK(List pindx)
{ List result = new Vector();
  for (int i = 0; i < pindx.size(); i++)
  { String pind = (String) pindx.get(i);
    result.add(personindex.get(pind));
  }
  return result;
}

to manage the indexing map.

Related patterns Used by phased creation to look up the target elements corresponding to previously-processed source model elements.

A.7 Recursive descent

Synopsis 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.
Forces Appropriate for relatively simple transformations in which there is a close structural correspondence between source and target models, and a strict hierarchy of entities (no cycles in the entity dependency relation).

Solution Decompose the transformation into operations, based upon the Cons constraints.

A constraint

$$\forall s : \text{SCond} \Rightarrow \exists t : \text{TCond} \text{ and Post}$$

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

Consequences The transformation implementation is tightly coupled to the source and target entity hierarchies, and cannot be decomposed into separate phases.

Code examples The implementation of the UML to relational database transformation in QVT or ATL are examples of this pattern [24].

A.8 Phased creation

Synopsis Construct target model elements in phases, ‘bottom-up’ from individual objects to composite structures, based upon a structural dependency ordering of the target language entities.

Forces Used whenever the target model is too complex to construct in a single step. In particular, if an entity depends upon itself via an association, or two or more entities are mutually dependent via associations. In such a case the entity instances are created first in one phase, then the links between the instances are established in a subsequent phase.

Solution Decompose the transformation into phases, based upon the Cons constraints. These constraints should be ordered so that data read in one constraint is not written by a subsequent constraint, in particular, phase \( p1 \) must precede phase \( p2 \) if it creates instances of an entity \( T1 \) which is read in \( p2 \).

Consequences The stepwise construction of the target model leads to a transformation implementation as a sequence of phases: earlier phases construct elements that are used in later phases. Some mechanism is required to look up target elements from earlier phases, such as by key-based search or by trace lookup.
Implementation. The constraints are analysed to determine the dependency ordering between the target language data and entities. \( T_1 < T_2 \) means that a \( T_1 \) instance is used in the construction of a \( T_2 \) instance. Usually this is because there is an association directed from \( T_2 \) to \( T_1 \), or because some feature of \( T_2 \) is derived from an expression using \( T_1 \) elements. Subclasses and superclasses of a given entity are considered as being equivalent for the purpose of defining \(<\), ie, it is actually an ordering between equivalence classes of entities.

If the order \(<\) is a partial order then the corresponding ordering of phases follows directly from \(<\): a phase that creates \( T_2 \) instances must follow all phases that create \( T_1 \) instances, where \( T_1 < T_2 \). However, if there are self-loops \( T_3 < T_3 \), or longer cycles of dependencies, then the phases creating the entities do not set the links between them, instead there must be a phase which follows all these phases which specifically sets the links.

Code examples. The ThreeCycle example illustrates the simple case. Here ThreeCycle \(<\) IntResult, so the phase implementing \( C_2 \) must follow that for \( C_1 \). Likewise in the tree to graph example, where Node \(<\) Edge.

A simple example of the second case of the pattern is shown in Figure 6.

![Diagram of mutually-dependent target entities example](image)

Fig. 6. Mutually-dependent target entities example

Here there are four constraints in Cons:

\[
\begin{align*}
\forall s : S_1 \cdot s.x > 0 & \implies \exists t : T_1 \cdot t.id2 = s.id1 \\
\forall s : S_1 \cdot s.x \leq 0 & \implies \exists t : T_2 \cdot t.id3 = s.id1 \\
\forall s : S_1 \cdot s.x > 0 & \implies T_1[s.id1].r2 = T_2[s.r.id1] \\
\forall s : S_1 \cdot s.x \leq 0 & \implies T_2[s.id1].r1 = T_1[s.r.id1]
\end{align*}
\]

This is a re-expression transformation, which loses some source model information (\( S_1 \) objects with \( x > 0 \) will only have a copy of \( r \rightarrow \text{select}(x \leq 0) \) in the target model, rather than the entire \( r \) set). \( T_1 \) and \( T_2 \) are mutually dependent. The first and second
constraints can be implemented by a single phase, this is then followed by a phase implementing the third and fourth constraints.

**Related patterns** Object indexing can be used to find the target elements constructed for particular source elements in earlier phases.

## B Write and read frames

The write frame \( \text{wr}(P) \) of a predicate is the set of features and classes that it modifies, when interpreted as an action (an action to establish \( P \)). This includes object creation. The read frame \( \text{rd}(P) \) is the set of classes and features read in \( P \). The read and write frames can help to distinguish different implementation strategies for conjunctive-implicative constraints. In some cases, a more precise analysis is necessary, where \( \text{wr}^*(P) \) and \( \text{rd}^*(P) \), which include the sets of objects written and read in \( P \), are used instead.

Table 2 gives the definition of some cases of these sets.

<table>
<thead>
<tr>
<th>( P )</th>
<th>( \text{rd}(P) )</th>
<th>( \text{wr}(P) )</th>
<th>( \text{rd}^*(P) )</th>
<th>( \text{wr}^*(P) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic expression ( e ) without quantifiers, logical operators or ( =, \in, \subseteq, [E] )</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td></td>
</tr>
<tr>
<td>( r ) multiple-valued ( e_1 \in [e_2] )</td>
<td>( \text{rd}(e_1) \cup \text{rd}(e_2) )</td>
<td>{}</td>
<td>( \text{rd}^<em>(e_1) \cup \text{rd}^</em>(e_2) )</td>
<td>( \text{rd}^*(e_2) \times {} )</td>
</tr>
<tr>
<td>( r ) multiple-valued ( e_1 = e_2 )</td>
<td>( \text{rd}(e_1) \cup \text{rd}(e_2) )</td>
<td>{}</td>
<td>( \text{rd}^<em>(e_1) \cup \text{rd}^</em>(e_2) )</td>
<td>( \text{rd}^*(e_1) \times {} )</td>
</tr>
<tr>
<td>( r ) multiple-valued ( e_1 \subseteq [e_2] )</td>
<td>( \text{rd}(e_1) \cup \text{rd}(e_2) )</td>
<td>{}</td>
<td>( \text{rd}^<em>(e_1) \cup \text{rd}^</em>(e_2) )</td>
<td>( \text{rd}^*(e_2) \times {} )</td>
</tr>
<tr>
<td>( E ) function ( e_1 )</td>
<td>( \text{rd}(e_1) \cup {E} )</td>
<td>{}</td>
<td>( \text{rd}^*(e_1) \cup {E} )</td>
<td>{}</td>
</tr>
<tr>
<td>( \forall x : E \cdot Q ) (in suceedent)</td>
<td>( \text{rd}(Q) \cup {E} )</td>
<td>( \text{wr}(Q) \cup {E} )</td>
<td>( \text{rd}^*(Q) )</td>
<td>( \text{wr}^*(Q) \cup {E} )</td>
</tr>
<tr>
<td>( \forall x : E \cdot Q ) (at outer level)</td>
<td>( \text{rd}(Q) \cup {E} )</td>
<td>( \text{wr}(Q) \cup {E} )</td>
<td>( \text{rd}^*(Q) \cup {E} )</td>
<td>( \text{wr}^*(Q) \cup {E} )</td>
</tr>
<tr>
<td>( Q ) implies ( Q )</td>
<td>( \text{rd}(Q) \cup \text{rd}(Q) )</td>
<td>( \text{wr}(Q) \cup \text{wr}(Q) )</td>
<td>( \text{rd}^<em>(Q) \cup \text{rd}^</em>(Q) )</td>
<td>( \text{wr}^<em>(Q) \cup \text{wr}^</em>(Q) )</td>
</tr>
<tr>
<td>( Q ) and ( R )</td>
<td>( \text{rd}(Q) \cup \text{rd}(R) )</td>
<td>( \text{wr}(Q) \cup \text{wr}(R) )</td>
<td>( \text{rd}^<em>(Q) \cup \text{rd}^</em>(R) )</td>
<td>( \text{wr}^<em>(Q) \cup \text{wr}^</em>(R) )</td>
</tr>
</tbody>
</table>

**Table 2.** Definition of read and write frames

In some cases, \( \text{wr}(Post) \cap \text{rd}(Post) \) may be non-empty, but \( \text{wr}^*(Post) \cap \text{rd}^*(Post) \) is empty. For example, the constraint

\[
\forall e_1 : \text{Edge}@\text{pre} \; ; \; e_2 : \text{Edge}@\text{pre} \;
\]

\[
e_1 \neq e_2 \; \text{and} \; e_1.\text{trg} = e_2.\text{src} \implies
\]

\[
\exists e : \text{Edge} \cdot e.\text{src} = e_1.\text{src} \; \text{and} \; e.\text{trg} = e_2.\text{trg}
\]

to compute the composition of edges in a graph. Here \( \text{trg} \) and \( \text{src} \) are both read and written to in the constraint. But \( \text{wr}^*(Post) \) is \( \{\} \times \{\text{src}, \text{trg}\} \), where \( e \in \text{Edge} - \text{Edge}@\text{pre} \) and so \( \text{wr}^* \) is disjoint for distinct applications of the constraint, and also
disjoint from $rd^*$ of the constraint, which has object set \( \{e_1, e_2\} \) where \( e_1 \in \text{Edge}_{\text{pre}} \) and \( e_2 \in \text{Edge}_{\text{pre}} \). Therefore approach 1 can be used to implement the constraint, instead of approach 3.