Int J Software Informatics, Volume 6, Issue 2 (2012), pp. 233–269
International Journal of Software and Informatics, ISSN 1673-7288
©2012 by ISCAS. All rights reserved.
E-mail: ijsi@iscas.ac.cn
http://www.ijsi.org
Tel: +86-10-62661040

Comparative Evaluation of Model Transformation Specification Approaches

Kevin Lano, Shekoufeh Kolahdouz-Rahimi, and Iman Poernomo

(Dept. of Informatics, King's College London, Strand, London, UK)

Abstract Model transformations have become a key element of model-driven software development, being used to transform platform-independent models to platform-specific models, to improve model quality, to introduce design patterns and refactorings, and to map models from one language to another. A large number of model transformation notations and tools exist. However, there are no guidelines on how to select appropriate notations for particular model transformation tasks, and no comprehensive comparisons of the relative merits of particular approaches. In this paper we provide a unified semantic treatment of model transformations, and show how correctness properties of model transformations can be defined using this semantics. We evaluate several approaches which have been developed for model transformation specification, with respect to their expressivity, complexity and support for verification, and make recommendations for resolving the outstanding problems concerning model transformation specification.

Key words: model transformations; model transformation specification; UML

Lano K, Kolahdouz-Rahimi S, Poernomo I. Comparative evaluation of model transformation specification approaches. *Int J Software Informatics*, **Vol.6, No.2 (2012): 233–269. http://www.ijsi.org/1673-7288/6/i120.htm**

1 Introduction

Model transformations are mappings of one or more software engineering models (*source* models) into one or more *target* models. The models considered may be graphically constructed using graphical languages such as the Unified Modelling Language (UML)[26], or can be textual notations such as programming languages or formal specification languages.

A large number of model transformation notations and tools have been defined, however there is no consensus on which kinds of model transformation specification are appropriate for which types of model transformation problem. In addition, tool support for demonstrating the correctness of model transformations remains limited: such correctness must consider whether the transformation is guaranteed to establish the constraints of the target model, and to preserve (possibly under interpretation) constraints of the source model, as well as the internal consistency of the transformation specification itself. Such verification processes require the existence of a sound semantics for model transformations, independent of any particular model transformation language.

Corresponding author: Kevin Lano, Email: kevin.lano@kcl.ac.uk
Received 2011-01-04; Revised 2011-06-25; Accepted 2011-09-26; Published online 2012-03-19.

Section 2 distinguishes different categories of model transformation, Section 3 defines a general semantics for model transformations and defines concepts of correctness for model transformations. Section 4 surveys techniques for the specification of transformations, and Sections 5, 6 and 7 apply several model transformation approaches on three transformation case studies. In Section 8 we give a systematic comparison of these transformation approaches. Finally in Section 9 we summarise our results.

2 Categories of Model Transformation

Semantically-based classifications of model transformation approaches were defined in Ref.[25], these considered:

- The languages the transformation operates upon: i.e., program-level versus model-level transformations and *endogenous* (source and target language are the same) versus *exogenous* (different source and target languages);

- *Horizontal* (transformation does not change abstraction level) versus *vertical* (source and target models are at different abstraction levels) transformation;

- The level of automation and complexity of the transformation, and the semantic correctness of the transformation.

Criteria for the effectiveness of a transformation language and tool were also proposed, including the ability to compose transformations and to demonstrate syntactic and semantic correctness.

In this section we use semantically-based criteria to classify transformations, in particular the criteria of languages used, of abstraction levels and of the semantic relation between the source and target models.

In this paper we will consider the following three general categories of model transformations:

Refinements These transformations refine models towards implementations. Examples include PIM to PSM transformations in the MDA, or code generation from PSMs. They may remove certain constructs or structures, such as multiple inheritance, from a model, and represent them instead by constructs which are available in a particular implementation platform. The semantics of the model may be changed, but all the properties of the original model should be true in the new model, via some interpretation. They are usually *exogenous*, and are essentially characterised by being *vertical* transformations (from a higher to a lower level), and by the target being semantically equivalent to or stronger than the source.

Quality improvements These transformations do not change the abstraction level of a model, and usually preserve its semantics (under a suitable interpretation) but improve its structure and organisation, eg., by factoring out duplicated elements. They normally operate on a single language. That is, they are endogenous, semantically preserving and horizontal transformations.

Re-expressions These translate a model in one language into its 'nearest equivalent' in a different language, such as a different version of the source language. This is useful for re-engineering, migration, validation and tool integration. These are exogenous, semantically preserving and horizontal transformations.

Within each category, further subcategories can be distinguished, for example *refactoring* is a particular subcategory of quality improvement transformation.

Mappings from one programming language to another would normally be re-expressions (for languages at the same level of abstraction), whilst transformation from a formal specification language to code would be a refinement.

3 Semantic Framework for Model Transformations

3.1 Metamodelling framework

We will consider transformations between languages specified using the Meta-Object Framework (MOF). Figure 1 shows the four-level metamodelling framework of UML using MOF. At each level, a model or structure can be considered to be an instance of a structure at the next higher level (MOF is an instance of itself).

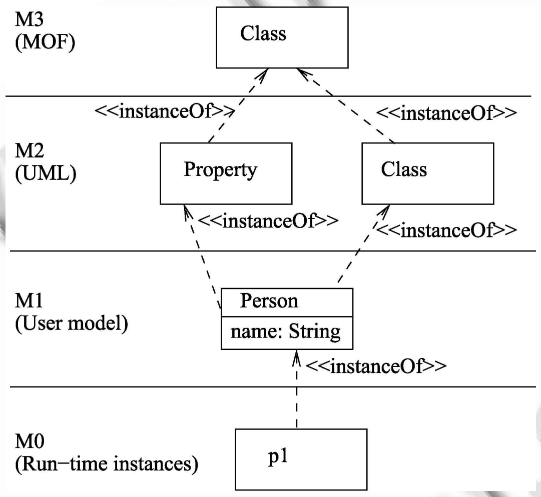


Figure 1. UML metamodel levels

Thus, for example, the concept of a class is defined in the UML metamodel (a model at level M2) by the metaclass *Class*, with associated collections of *ownedAttributes*, *ownedOperations* and other features (Fig. 2). Actual user models (class diagrams) are instances of this metamodel, and exist at level M1. In discussing model transformation correctness, we will refer to the M2 level as the *language level* (the models define the languages which the transformation relates) and to the M1 level as the *model level* (the models which the transformation operates upon).

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 Γ_M in $\mathcal{L}_M$, which defines the semantic meaning of M, including any internal constraints of M. These can also be defined for M1 models which are themselves UML models such as particular class diagrams.

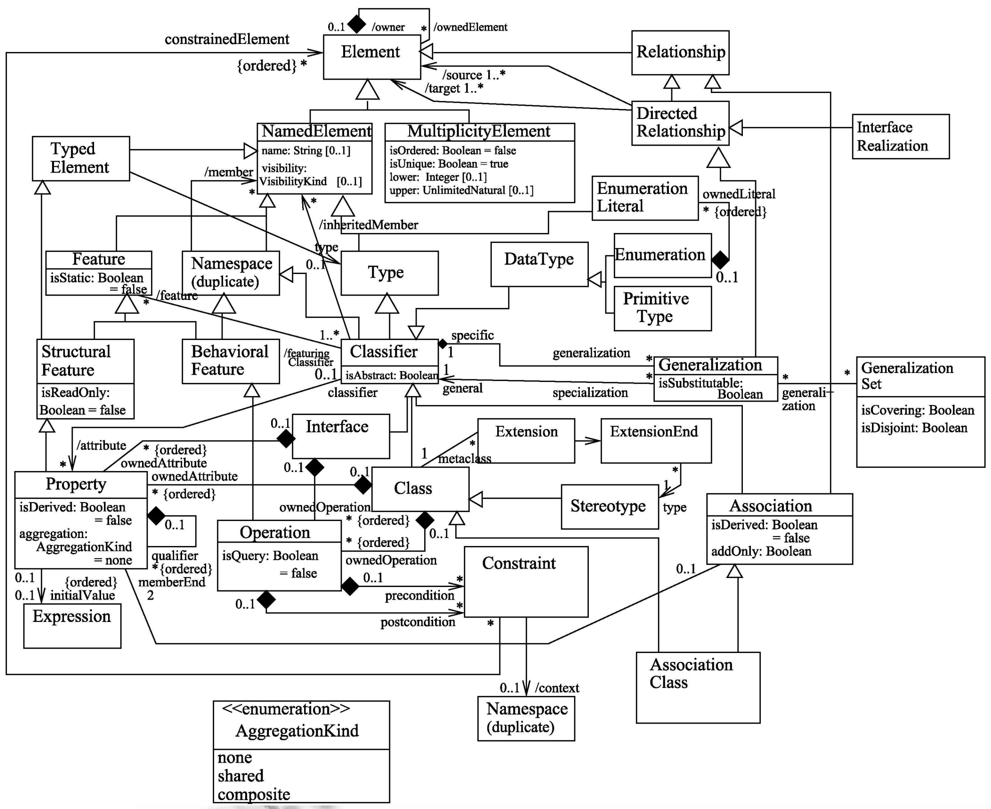


Figure 2. UML class diagram metamodel

The first-order language $\mathcal{L}_M$ 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 semantic types C for each classifier C defined in M. There are attribute symbols $\mathsf{att(c : C)}$: T for each property *att* of type T in the feature set of a classifier C. There are attributes $\overline{\mathsf{C}}$ to denote the set of currently existing instances of each classifier C, ie, its extent. This corresponds to $C.allInstances()$ in OCL. There are action symbols $\mathsf{op(c : C, p : P)}$ for each operation $op(p : P)$ in the features of C[18]. Collection types and operations on these and the primitive types are also usually included.

The theory Γ_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, including pre/post specifications of operations. For example, if classifier C generalises classifier D, this is expressed by the axiom $\overline{\mathsf{D}} \subseteq \overline{\mathsf{C}}$.

For a sentence φ 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.

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 Γ_L, although these cannot be expressed within $\mathcal{L}_M$ itself. We use the notation $M \models \varphi$ to express satisfaction of an $\mathcal{L}_L$ sentence φ in M.

For example, any particular UML class diagram satisfies the language-level property that there are no cycles in the inheritance hierarchy.

3.2 Model transformation semantics

Transformations can be regarded as mappings or relations between models. These models may be in the same or in different modelling languages. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the languages concerned. We assume these are defined as metamodels using MOF. A transformation τ then describes which models M_1 of $\mathcal{L}_1$ correspond to (transform to) which models M_2 of $\mathcal{L}_2$.

Let $\mathit{Models}_{\mathcal{L}}$ be the set of models which interpret the language (metamodel) $\mathcal{L}$. We may simply write $M : \mathcal{L}$ instead of $M : \mathit{Models}_{\mathcal{L}}$.

A model transformation τ from language $\mathcal{L}_1$ to language $\mathcal{L}_2$ can therefore be expressed as a relation

$$\mathit{Rel}_\tau : \mathit{Models}_{\mathcal{L}_1} \leftrightarrow \mathit{Models}_{\mathcal{L}_2}$$

Sequential composition $\tau;\ \sigma$ of transformations corresponds to relational composition of their representing relations.

In general it may be that only some models in $\mathcal{L}_1$ can have a transformation τ validly applied to them: this is termed the *applicability condition* of τ. It is defined as membership of the domain of Rel_τ:

$$\mathrm{dom}(\mathit{Rel}_\tau) = \{M : \mathit{Models}_{\mathcal{L}_1} \mid \exists\, M' : \mathit{Models}_{\mathcal{L}_2} \cdot \mathit{Rel}_\tau(M, M')\}$$

A transformation is *invertible* if it Can be applied in the reverse direction. The reversed transformation τ^{-1} is represented by the inverse Rel_τ^{-1} relation, which is only defined on the domain

$$\mathrm{ran}(\mathit{Rel}_\tau) = \{M : \mathit{Models}_{\mathcal{L}_2} \mid \exists\, M' : \mathit{Models}_{\mathcal{L}_1} \cdot \mathit{Rel}_\tau(M', M)\}$$

The reversed relation may not be functional, since there may be different models of $\mathcal{L}_1$ which map to the same model of $\mathcal{L}_2$.

A model transformation implementation is said to be *change propagating* if changes Δs to the source model s can be used to compute a necessary change Δt to the target model, without the need to re-execute the transformation on the entire modified source model $s \oplus \Delta s$[32].

If two transformations τ and σ have change-propagating implementations, so does $\tau;\ \sigma$.

The formalism we have introduced here permits ternary or higher-multiplicity transformations, for example a transformation

$$\mathit{union} : \mathcal{L} \times \mathcal{L} \leftrightarrow \mathcal{L}$$

which produces a union of two models, can be considered as a relation relating pairs of source models to single target models. In this paper we will only consider binary transformations.

3.3 Model transformation correctness

The following notions of transformation correctness have been defined[33, 3]:

- *Syntactic correctness*: the transformation always produces syntactically well-formed models of the target language from valid models of the source language. This also requires that every semantic constraint of the target language is satisfied by the target model.

- *Definedness*: the transformation is validly defined on each source model.

- *Uniqueness*: the transformation produces a unique result from a given starting model.

- *Completeness*: for individual rules this means that all aspects of the target model which cannot be automatically inferred (such as default initial values for properties) are explicitly specified by the rule.

  For the entire transformation it means that each expected transformation relationship between a pair of models can actually be established by some composition of transformation rules of the transformation.

- *Semantic correctness*: for each property of the source model which should be preserved (correctness properties), the target model satisfies the property, under a fixed interpretation of the source language into the target language.

  This divides into *language-level semantic correctness*, for properties expressed in the source language itself, and *model-level semantic correctness*, for properties expressed in the language of an individual source model (such as constraints within the model). Different interpretations usually apply at these different levels.

The first three properties are termed *Strong executability*, *Totality* and *Determinism* in Ref. [3].

In our semantics for transformations, we can precisely define these criteria as follows for a model transformation τ from $\mathcal{L}_1$ to $\mathcal{L}_2$:

Syntactic correctness For each model which conforms to (is a model in the language) $\mathcal{L}_1$, and to which the transformation can be applied, the transformed model conforms to $\mathcal{L}_2$:

$$\forall\, M_1 : \mathcal{L}_1;\ M_2 \cdot Rel_\tau(M_1, M_2)\ \Rightarrow\ M_2 : \mathcal{L}_2$$

Definedness This means that the applicability condition of Rel_τ is *true*: its domain is the complete collection of models of $\mathcal{L}_1$.

Uniqueness This means that Rel_τ is functional as a relation from $\mathcal{L}_1$ to $\mathcal{L}_2$.

Completeness If M_1 should be related to M_2 by the transformation, then $Rel_\tau(M_1, M_2)$ holds.

Semantic correctness 1. Language-level correctness: each language-level property $\varphi : \mathcal{L}_{\mathcal{L}_1}$ satisfied by M_1 is also satisfied, under an interpretation χ on language-level expressions, in M_2:

$$\forall\, M_1 : \mathcal{L}_1;\; M_2 : \mathcal{L}_2 \cdot Rel_\tau(M_1, M_2) \;\wedge\; M_1 \models \varphi \;\Rightarrow\; M_2 \models \chi(\varphi)$$

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 ζ on model-level expressions, in the corresponding target model M_2:

$$\forall\, M_1 : \mathcal{L}_1;\; M_2 : \mathcal{L}_2 \cdot Rel_\tau(M_1, M_2) \;\wedge\; \Gamma_{M_1} \vdash \varphi \;\Rightarrow\; \Gamma_{M_2} \vdash \zeta(\varphi)$$

It is sufficient to consider $\varphi \in \Gamma_{M_1}$ in this case.

Model-level semantic correctness should be expected for refinement and quality improvement transformations. For re-expression transformations there may be cases where M_1 properties cannot be expressed in M_2 (ζ will be a partial interpretation), but all expressible properties should be preserved from M_1 to M_2. The interpretations can also be used to map test cases θ of M_1 into test cases $\zeta(\theta)$ of M_2.

Some transformations may be inherently non-functional, for example, the transformation (translation) between the abstract syntax trees representing texts in different natural languages. However, as Ref. [3] points out, non-determinism can also result from specification errors such as under-specification and omission of intended predicates.

If transformations τ and σ are semantically correct at a particular level, so is their composition $\tau;\ \sigma$, using the composition of the interpretations χ_τ, χ_σ, or ζ_τ, ζ_σ.

4 Specification Techniques for Model Transformations

A large number of formalisms have been proposed for the definition of model transformations: the *pure relational* approach of Refs. [1, 2], constructive type theory[30], *graphical* description languages such as graph grammars[6,31], or the visual notation of QVT[27], hybrid approaches such as Epsilon[14] and implementation-oriented languages such as Kermeta[11].

The key problem with the specification of model transformations is that, semantically, model transformations are relations from one (or more) entire model to one or more entire models, but that such a global description is impractical for non-trivial languages and transformations. Instead, model transformations are specified in terms of relations between individual elements in the source model(s) and individual elements in the target model(s). The model-to-model relation is then derived from some composition of these individual relations. This gives rise to potential ambiguity and incompleteness due to possible alternative compositions.

A related problem is that languages such as UML are defined by highly complex metamodels, in which meta-entities depend on each other. A transformation which operates on some c : *Class* in the source model may need to also transform the *Property* elements which are owned attributes of C, the *Operation*s which are owned operations of C, and so forth. In terms of transformation rules, a rule for *Class* will typically depend on rules for *Property* and *Operation*, which may in turn depend on rules for other metaclasses, including superclasses of *Class*, such as *Type*.

Ideally, any specification language for model transformations should support validation, modularity, verification, and the implementation of transformations:

Modularity It is possible to organise model transformation rules into modules, which have high internal cohesion and low coupling. Model transformations may be composed *externally*[12] as complete modules, or rules within a transformation may be considered to be modules, and composed *internally* within a transformation. Finally, *intra-rule* composition allows individual rules to be decomposed into parts.

Validation It is possible to analyse the specification to ensure it represents the correct intended transformation. This might be done by inspection, animation, testing, etc.

Verification It should be possible to prove that a transformation is consistent, satisfiable, and semantically correct, ie., that all constraints of the source model remain true in the target model (possibly under some interpretation).

Implementation The specification can be used to automatically generate an efficiently executable implementation of the transformation, which is correct with respect to the specification.

Modularity is the key property which supports the other three properties. Transformations described in a monolithic manner, or as an unstructured set of rules cannot be easily understood, analysed or implemented. Instead, if transformations can be decomposed into appropriate smaller units, such as coherent groups of closely-related rules, these parts can be (in principle) more easily analysed and implemented, and the verification and implementation of the complete transformation can be composed from those of its parts.

Suitable modularity mechanisms also improve the flexibility and reusability of a transformation, making it easier to modify or reuse parts of the transformation independently of other parts.

Three general styles of specification have been used for defining model transformations:

Declarative Transformations are described abstractly, eg., as mathematical relations between source and target models[1,2,30].

Imperative Transformations are defined as programs which explicitly define the details of how a source model is transformed into a target model[11].

Hybrid A combination of declarative and imperative, eg., a wide-spectrum specification language in which a declarative description can be refined within the same notation into a program-like description[14].

The declarative style has the advantage that (in principle), transformations can be described more clearly and concisely, omitting the details of strategies for selecting and modifying model elements, and avoiding explicitly ordering the application of rules on specific elements. The imperative style on the other hand makes it easier and more direct to implement the transformation in an executable form. The hybrid style attempts to combine the other two styles in order to obtain the advantages of both.

We will compare five transformation specification approaches on three transformation problems:

1. A re-expression transformation from trees to graphs;
2. A refinement transformation from UML to relational database schemas;
3. A quality improvement transformation on class diagrams, to factor out duplicated attributes within a class hierarchy.

The approaches chosen represent declarative (QVT-Relations, ATL), graph-transformation (VIATRA), hybrid (UML-RSDS) and imperative (Kermeta) specification approaches.

We will analyse the approaches with regard to their modularity, validation, verification, complexity, interoperability, usability, the ability to produce executable implementations, and other desirable properties of a model transformation formalism.

The general properties we will consider are:

Level of abstraction The degree to which the specification can abstract from implementation details. A highly abstract notation is usually more concise and closer to requirements, and so easier to validate. However it Can be inefficient to implement.

Modularity How the specification can be organised and structured.

Complexity This will be measured by metrics on the case studies, such as counts of operation calls and call depths.

Interoperability Whether the approach can be used conveniently within a software development process in combination with other development tools.

Usability The developer effort required to construct and analyse the specification.

Implementability The capability to generate executable implementations of the transformation.

We will consider also the support provided for showing syntactic correctness, completeness, definedness and confluence, and the support for bidirectional and change-propagating transformations.

We will evaluate the ability of the approaches to support a range of different specification styles, and evaluate their appropriateness for the different types of transformation (re-expressions, refinements, quality improvements) considered here.

5 Case Study 1: Tree to Graph Transformation

Figure 3 shows the source and target metamodels of the transformation from trees to graphs. The *identity* constraint in the metamodel means that tree nodes must have unique names, and likewise for graph nodes.

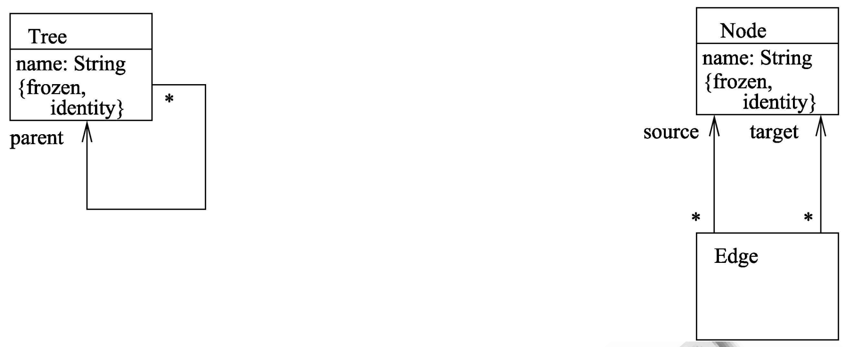


Figure 3. Tree to graph transformation metamodels

We can express the requirements of the transformation in OCL, using a conventional mathematical syntax for the logical operators. The tree metamodel has the language constraint that there are no non-trivial cycles in the *parent* relationship ($Asm1$):

$$t : Tree \ and \ t \neq t.parent \quad implies \quad t \notin t.parent^{+}$$

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.

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

$$e : Edge \quad implies \quad e.source \neq e.target$$

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

$$\begin{array}{l} e1 : Edge \ and \ e2 : Edge \ and \\ e1.source = e2.source \ and \ e1.target = e2.target \ implies \ e1 = e2 \end{array}$$

These constraints must therefore be established by any syntactically correct transformation to graphs.

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.

We can formally specify the transformation by four global constraints:

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

$$t : Tree \ implies \ \exists \, n : Node \cdot n.name = t.name$$

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

$$\begin{array}{l} t : Tree \ and \ t.parent \neq t \quad implies \\ \exists_1 \, e : Edge \cdot e.source.name = t.name \ and \ e.target.name = t.parent.name \end{array}$$

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

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

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

$$e : Edge\ implies\ \exists_1\, t : Tree \cdot t.parent \neq t\ and$$
$$t.name = e.source.name\ and\ t.parent.name = e.target.name$$

$C3$ and $C4$ are duals of $C1$ and $C2$, defining a reverse direction, from graphs to trees, of the transformation, so that it is (in principle) bidirectional.

We will compare the specifications of this transformation in five different formalisms: QVT-Relations[27], ATL[8], VIATRA[29], UML-RSDS[20] and Kermeta[11].

5.1 QVT-relations

The QVT-Relations (QVT-R) formalism is a notation for defining model transformations as sets of rules. Each rule consists of left and right-hand side object models showing generic structures of instances from the source and target metamodels. There is also a *when* clause, specified in OCL, which defines necessary assumptions or preconditions for the valid application of the rule, and a *where* clause, defining effects (postconditions) that the rule should establish when it completes. Graphical notation can be used for rules, here we will use the equivalent textual notation of QVT-R.

The tree to graph transformation can be defined by three QVT rules *Tree2NodeMain*, *Tree2Node* and *Tree2Edge*:

```
transformation simpleExample(source : Tree, target : Node) {

top relation Tree2NodeMain {
  tn : String;

  checkonly domain source s : Tree::Treeclass { tname = tn };
  enforce domain target t : Node::Nodeclass { nname = tn };
  when { s.parent.oclIsUndefined(); }
}

relation Tree2Node {
  tn : String;

  checkonly domain source s : Tree::Treeclass { tname = tn };
  enforce domain target t : Node::Nodeclass { nname = tn };
}

top relation Tree2Edge {
   checkonly domain source s : Tree::Treeclass {};
   enforce domain target e : Node::Edgeclass {};
   when { not s.parent.oclIsUndefined(); }
   where { Tree2Node(s, e.sourceref);
           Tree2Node(s.parent, e.targetref); }
```

```
    }
}
```

It was necessary to modify the source metamodel, since the tool used (Medini QVT) did not accept the 1-multiplicity *parent* role. Instead this was made into a 0..1 role, and the test *s.parent.oclIsUndefined*() identifies if the tree *s* is a root. Separate name attributes *tname* and *nname* have been defined for trees and nodes.

The specification is structured as a set of rules, called *relations*, which operate upon elements termed *domain*s, in this case *source* and *target* domains. The notation *enforce* means that the rule is enforced by modifying a domain (such as the source or target models) where necessary. *checkonly* indicates that the domain is checked but not modified by the rule. It is possible for both source and target to be *enforce*, supporting bidirectional transformations, although there are semantic problems with such transformations[32]. A *top* relation is executed upon all elements in the source model to which it is applicable, in an arbitrary order, other relations are invoked (directly or indirectly) from top relations.

In this example the rule *Tree*2*Edge* maps trees with parents to edges, the source and target nodes of these edges are recursively created by invoking *Tree*2*Node* in the *where* clause. The rule *Tree*2*NodeMain* maps trees without parents to nodes.

This strategy could be termed *recursive descent*: that is, the construction of target model elements involves the recursive construction of their sub-parts, with the recursion terminating at basic elements without sub-parts (in this example, *Node* elements). This is the usual strategy used with QVT-R.

5.2 ATL

The ATLAS Transformation Language (ATL) is a declarative model transformation language. An ATL specification consists of a set of rules, each rule has a source and target pattern, specified using OCL. A rule application occurs when the source pattern of the rule matches some part of the source model, corresponding elements that satisfy the target pattern are then created in the target model. A rule may implicitly call another rule, or inherit from another rule, and may also explicitly invoke other rules (similarly to the invocation of relations within the *where* clause of a QVT relation).

The tree to graph transformation can be coded as follows in ATL:

```
module Tree2Node; -- Module Template create OUT : NodeMM from IN :
TreeMM ;

rule Tree2Node { from
  t : TreeMM!Tree(t.parent->noEmpty() and not t.parent=t)
to
  out : NodeMM!Node(
    name <- t.name
  ),
  edg : NodeMM!Edge(
    source <- out,
    target <- t.parent
  )
}
```

```
rule Tree2NodeSecond { from
  t : TreeMM!Tree(not t.parent->noEmpty() or t.parent=t)
to
  out : NodeMM!Node(
    name <- t.name
  )
}
```

The first rule processes tree nodes with parents (distinct from the node), and creates both a new graph node for the tree, and an edge connecting it to (the node corresponding to) its parent. The second rule processes tree nodes without parents, and creates only a new corresponding graph node. When setting *edg.target*, the first rule implicitly depends upon itself or the second rule to map *t.parent* to a graph node which can be assigned to this reference. Again, we needed to modify the source metamodel, to make *parent* optional.

5.3 VIATRA

VIATRA[29] is a graph transformation based model transformation language. In VIATRA, transformation rules are specified by graph matching and graph rewriting: a pattern language is used to select elements from the model, and a rule defines how these and related elements are modified. A procedural language, ASM, is used to explicitly schedule the application of rules using sequencing, loops, conditionals, etc, so that VIATRA is also a hybrid specification language. Unlike the other approaches discussed here, VIATRA is not based upon MOF and OCL, but uses a different metamodelling formalism, VPM[29]. VIATRA is a general framework which supports the use of many different modelling languages. The notation used for transformation rules is related to a logic-programming formalism such as Prolog, instead of being based on OCL.

The tree to graph transformation can be specified in VIATRA as a *machine* consisting of two graph transformation rules, each defined by a pre and postcondition pattern, and a top-level algorithm to control the order of their execution:

```
machine tree2graph { gtrule mapTreeToNode(in T) =
  { precondition pattern isTree(T,Nme) =
    { tree(T); name(T,Nme); }
    postcondition pattern newNode(N,Nme) =
    { node(N); name(N,Nme); }
  }

  gtrule mapTreeToEdge(in T) =
  { precondition pattern isTreeWithParent(T,N,P,N1) =
    { tree(T); parent(T,P); tree(P); check(T != P);
      name(T,Nme1); node(N); name(N,Nme1);
      name(P,Nme2); node(N1); name(N1,Nme2);
    }
    postcondition pattern newEdge(N,N1,E) =
    { edge(E); source(E,N); target(E,N1); }
  }
  rule main() =
  seq {
    forall T with apply mapTreeToNode(T) do skip;
    forall T with apply mapTreeToEdge(T) do skip;
  }
}
```

The specification uses a phased approach to produce the target model: basic elements of the target model are produced in the initial phases of the transformation, then elements that are constructed from these elements are produced in successive phases, and so forth.

5.4 UML-RSDS

UML-RSDS is a model-driven development approach which generates executable systems from high-level specifications consisting of class diagrams, constraints and state machines[17]. It can be used to specify model transformations by formalising them as constraints or as operations at the metamodel level. Code generation can then be used to produce executable implementations of the model transformations.

UML-RSDS defines transformations as particular use cases of a system, and the behaviour of these use cases is abstractly specified by sets of constraints. From the constraints, explicit transformation rules as operations can be automatically derived.

In this example, the mapping from trees to graphs could be expressed by two constraints of a use case *tree2graph*, with both constraints having context *Tree*:

$$\forall\, t : \mathit{Tree} \cdot \exists\, n : \mathit{Node} \cdot n.\mathit{name} = t.\mathit{name}$$

and

$$\begin{array}{l}\forall\, t : \mathit{Tree} \cdot t.\mathit{parent} \neq t \quad \mathit{implies} \\ \exists_1\, e : \mathit{Edge} \cdot e.\mathit{source} = \mathit{Node}[t.\mathit{name}] \ \mathit{and} \ e.\mathit{target} = \mathit{Node}[t.\mathit{parent}.\mathit{name}]\end{array}$$

The notation $\mathit{Node}[x]$ refers to the node object with primary key (in this case name) value equal to x, it is implemented in the UML-RSDS tools by maintaining a map from the key values to nodes. In OCL it would be expressed as

$$\mathit{Node}.\mathit{allInstances}()\rightarrow \mathit{select}(\mathit{name} \ = \ x)\rightarrow \mathit{any}()$$

No changes were needed to the source or target metamodels. UML-RSDS also supports the definition of metamodel constraints, such as the no-cycles property of the tree metamodel.

In contrast to QVT-R and ATL, where the pre and postcondition predicates of rules simply specify the effect of one step within a transformation, the above constraints are intended to hold at the termination of the complete transformation (for all tree objects). They are therefore at a higher level of abstraction, since they are independent of particular strategies for executing the transformation.

However, they can also be interpreted in an operational manner to define individual transformation steps within a transformation. From the constraints, the following operations of *Tree* are derived to carry out such steps:

$\mathit{mapTreeToNode}()$

```
post:
  $Node \rightarrow exists( \ n \ \mid \ n.name \ = \ name \ )$


$mapTreeToEdge()$
post:
  $self \ \neq \ parent \quad implies \quad Edge \rightarrow exists1( \ e \ \mid$
      $e.source \ = \ Node[name] \quad and$
      $e.target \ = \ Node[parent.name] \ )$
```

$E \rightarrow exists(x \mid P)$ is the conventional OCL syntax for $\exists\, x : E \cdot P$.

These operations are executed using the phased strategy: the tree-to-node mappings are iterated first, then the tree-to-edge mappings:

```
  for $t$ : $Tree$ do $t.mapTreeToNode()$ ;
  for $t$ : $Tree$ do $t.mapTreeToEdge()$
```

The operations and activity are generated automatically from the constraints, together with executable Java code, which is correct by construction with respect to the constraints.

Likewise in the graph to tree direction, a similar use case specification can be defined, consisting of constraints for $C3$ and $C4$, and derived operations *mapNodeToTree*() on *Node* and *mapEdgeToTree*() on *Edge* which define a tree and its *parent* association from a graph.

5.5 Kermeta

Kermeta is a Java-like object-oriented programming language, constructed on the type system of the OMG Meta Object Framework (MOF)[28], the base UML notation in which UML metamodels are expressed. Kermeta can be used to define metamodels, as sets of classes, and to transform instances (models) of these metamodels.

Several different styles of transformation specification are possible with Kermeta. We could define, within each source language metaclass, operations to specify the transformation rule(s) applicable to this type of source model element. Alternatively, a separate transformation metaclass can be defined, independent of the source or target metamodels:

```
class Tree2Graph {
  operation createNodes(inputParent:TMM1, inputroot:Root) : NMM2 is do
    var nodeForParent : NMM2 init NMM2.new
    nodeForParent.name := inputParent.name
    inputroot.Nodes.add(nodeForParent)

    inputParent.child.each{ t |
      stdio.writeln("Now create Node for child " + t.name)
      var nodeForChild := createNodes(t,inputroot)
      var newEdge : EdgeMM init EdgeMM.new
      newEdge.source := nodeForChild
      newEdge.target := nodeForParent
      inputroot.Edges.add(newEdge)
    }
    result := nodeForParent
  end

  operation main() : Void is do
    var root : Root init Root.new
    var inputModel : TMM1 init loadTMM1()
    createNodes(inputModel,root)
    var repository : EMFRepository init EMFRepository.new
    var resource : Resource init repository.createResource(
      "platform:/resource/models/NMM2.xmi",
      "platform:/resource/metamodel/NodeMM2.ecore")
    resource.instances.add(root)
    resource.save()
  end
}
```

The operation *createNodes* applies a recursion down the structure of a tree, from the parent to its children, to map trees into nodes and edges. Such operations can be iterated over all the *Tree* elements in a set by using an iterator operator such as `each`:

```
  inputParent.child.each { t | createNodes(t, root) }
```

Other OCL collection operators such as *forAll*, *select* and *collect* also have implementations in Kermeta.

A phased strategy could alternatively have been defined. As with QVT and ATL, it was necessary to alter the source metamodel, in this case to add the inverse role *child* of *parent*.

6 Case Study 2: UML to Relational Database Mapping

This case study concerns the mapping of a data model expressed in UML class diagram notation to the more restricted data modelling language of relational database schemas. Modelling aspects such as inheritance, association classes, many-many associations and qualified associations need to be removed from the source model and their semantics expressed instead using the language facilities (tables, primary keys and foreign keys) of relational databases.

Figure 4 shows the source and target metamodels (as used in the UML-RSDS version of the specification, these are also close to the metamodels used in QVT-R).

We consider the published specifications of this problem for QVT-R[27], ATL[9], VIATRA[29], UML-RSDS[23] and Kermeta[5].

6.1 QVT-relations

The specification uses a recursive descent strategy, combined with implicit ordering of relations. There are three top relations:

1. $PackageToSchema(p, s)$

2. $ClassToTable(c, t)$

3. $AssocToFKey(a, fk)$

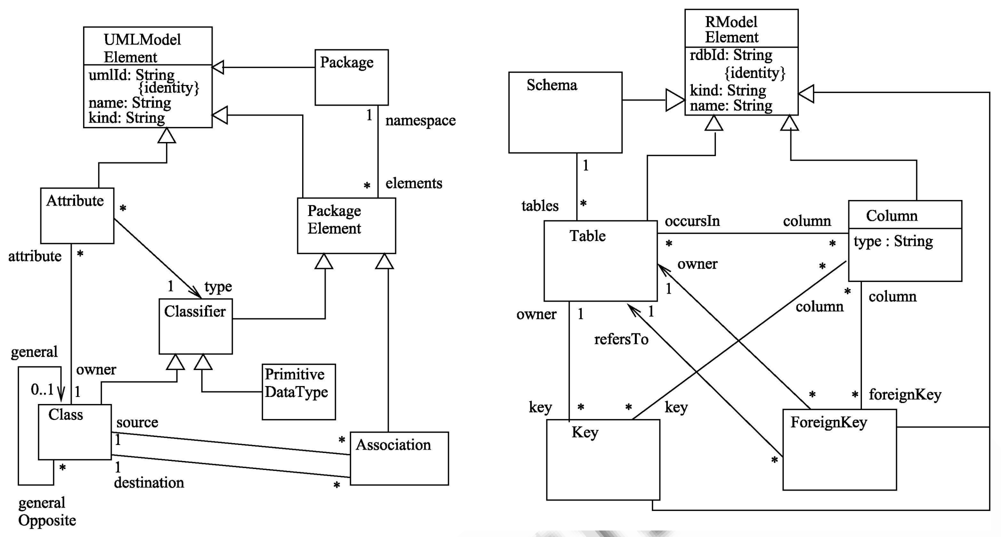


Figure 4. UML to relational database transformation metamodels

These are (loosely) ordered in this order: for a specific class, its package must already be mapped to a schema before the class can be mapped to a table, due to the *when* clause in the definition of *ClassToTable*:

```
top relation ClassToTable { cn, prefix : String;

  checkonly domain uml c : Class { namespace = p : Package {},
          kind = 'Persistent', name = cn };
  enforce domain rbdms t : Table { schema = s : Schema {},
          name = cn,
          column = cl : Column { name = cn + '_tid', type = 'NUMBER' },
          key = k : Key { name = cn + '_pk', column = cl } };
  when
  { PackageToSchema(p,s); }
  where
  { prefix = '';
    AttributeToColumn(c,t,prefix);
  }
}
```

In turn, *AssocToFKey*(a, fk) requires that the classes at either end of association a have been mapped to tables by *ClassToTable* before the association can be mapped. However the rules can be otherwise applied in any order.

Classes are mapped to tables by mapping their individual attributes, which is achieved by the relation *AttributeToColumn*:

```
relation AttributeToColumn { checkonly domain uml c : Class {};
  enforce domain rdbms t : Table {};
  primitive domain prefix : String;
  where
  { PrimitiveAttributeToColumn(c,t,prefix);
    ComplexAttributeToColumn(c,t,prefix);
    SuperAttributeToColumn(c,t,prefix);
  }
}
```

In the *where* clause a *disjunction* of the three relations for primitive, complex and supertype attributes is invoked, although this is only identifiable by examining the disjoint matching conditions of these sub-relations: the same *where* syntax is used for conjunction of invoked relations.

Table 3 shows the call graph of the specification.

6.2 ATL

The specification of the problem in ATL uses a similar recursive descent approach to the QVT solution.

```
rule Class2Table { from  c : Class!Class
  to
    out : Relational!Table (
        name <- c.name,
        col <- Sequence {key}->union(c.attr->select( e | not e.multiValued)),
        key <- Set {key}
    ),
    key : Relational!Column (
        name <- 'objectId',
        type <- thisModule.objectIdType
    )
}
```

The rule creates a table *out* with name set to *C*'s name, and columns a sequence of columns starting with the *key* column, followed by the columns created by implicit calls of some rule to map an attribute to a column (as in the QVT example, an implicit disjunction of such rules is also being used here). The key of the table is the single *key* column created in the rule, it has a default type, in this case *Integer*.

The four rules for mapping attributes to columns have disjoint preconditions, to cover the cases:

- *DataTypeAttribute*2*Column*: the attribute is of a data type (not a class type) and is not multivalued;
- *MultiValuedDataTypeAttribute*2*Column*: for multivalued data type attributes;
- *ClassAttribute*2*Column*: for single-valued attributes of a class type (ie., 1-multiplicity associations to the class of the type);
- *MultiValuedClassAttribute*2*Column*: for multi-valued attributes of class type, ie, *-multiplicity associations.

The first case has the form:

```
rule DataTypeAttribute2Column { from
    a : Class!Attribute
    ( a.type.oclIsKindOf(Class!DataType) and not a.multiValued )
  to
```

```
    out : Relational!Column
    ( name <- a.name,
      type <- a.type
    )
}
```

The transformation does not consider source models with inheritance.

6.3 VIATRA

The VIATRA specification of this transformation consists of a *main* rule, which iterates individual rules in a phased construction of the model, from the top down, and the rules defining the mapping of individual source model elements.

For example, the rule for mapping classes to tables is:

```
gtrule class2tableR(in Cls) = { precondition pattern lhs (Cls,
ClsNM) =
  { Class(Cls) below models("uml");
    NamedElement.name(N1, Cls, ClsNM);
    String(ClsNM) below models("uml");
  }
  action
  { let T = undef in
    let R = undef in
    let RS = undef in
    let RT = undef in seq
    { call createNewTable(value(ClsNM), T);
      call createPrimaryKeyInTable(T);
      new (class2table(R) in models("ref"));
      new (class2table.srcRef(RS,R,Cls));
      new (class2table.trgRef(RS,R,T));
      print("Class " + fqn(Cls) + "-> Table" + fqn(T) + "\n");
    }
  }
```

In contrast to the purely declarative rules *mapTreeToNode* and *mapTreeToEdge* of the tree to graph case study, the rule here is defined operationally with a sequentially-executed *action* instead of a postcondition. The first two calls in this action invoke subordinate rules, the remaining statements create trace information.

The trace is used to look up previously created target data from its corresponding source data, to support the phased specification strategy. For example the subsequent rule $attr2columnR(in\ Cls, in\ Attr)$ uses the precondition predicates

```
  class2table.srcRef(RS, R, Cls);
  class2table(R) below models("ref");
  class2table.trgRef(RS, R, Tab);
```

to find the table generated from *Cls*.

The overall algorithm is defined in the *main* rule:

```
rule main (in UMLStr, in RefStr, in DBStr) = seq { call
initModels(UMLStr, RefStr, DBStr);
```

```
  forall C below models("uml") with apply class2tableR(C) do skip;
  forall C below models("uml"), A below models("uml")
    with apply attribute2columnR(C,A) do skip;
  forall A below models("uml") with apply attrOfStringTypeR(A) do skip;
  forall A below models("uml") with apply attrOfIntTypeR(A) do skip;
  forall A below models("uml") with apply assoc2tableR(A) do skip;
}
```

The transformation does not consider source models with inheritance, but a code generator to produce textual schema definition code from the target model is provided.

6.4 UML-RSDS

In UML-RSDS the formal specification of the transformation as a single global relation between the source and target languages is split into six core constraints, for example:

C1 "For each persistent attribute in the source model there is a unique column in the target model, of corresponding type":

$$\begin{array}{l} \forall\, a : Attribute \;\cdot\; a.owner.kind = `Persistent' \; implies \\ \quad \exists_1\, cl : Column \cdot cl.rdbId = a.umlId \; and \\ \quad cl.name = a.name \; and \; cl.kind = a.kind \; and \\ \quad (a.type.name = `INTEGER' \; implies \; cl.type = `NUMBER') \; and \\ \quad (a.type.name = `BOOLEAN' \; implies \; cl.type = `BOOLEAN') \; and \\ \quad (a.type.name \neq `INTEGER' \; and \; a.type.name \neq `BOOLEAN' \; implies \\ \qquad cl.type = `VARCHAR') \end{array}$$

C2 "For each persistent class in the source model, there is a unique table representing the class in the target model, with columns for each owned attribute":

$$\begin{array}{l} \forall\, c : Class \;\cdot\; c.kind = `Persistent' \; implies \\ \qquad \exists_1\, t : Table \cdot t.rdbId = c.umlId \; and \; t.name = c.name \; and \\ \qquad\qquad t.kind = `Persistent' \; and \\ \qquad\qquad Column[c.attribute.umlId] \subseteq t.column \end{array}$$

Here the lookup of previously processed elements is carried out by a search based on primary key values, to find the columns derived from *c.attribute*.

C3 "For each root class in the source model there is a unique primary key in the target model":

$$\begin{array}{l} \forall\, c : Class \;\cdot\; c.kind = `Persistent' \; and \; c.general = \{\} \; implies \\ \exists_1\, k : Key \cdot k.rdbId = c.umlId + ``\_Pk" \; and \; k.name = c.name + ``\_Pk" \; and \\ \qquad k.owner = Table[c.umlId] \; and \; k.kind = `PrimaryKey' \; and \\ \qquad \exists_1\, cl : Column \cdot cl.rdbId = c.umlId + ``\_Id" \; and \\ \qquad\quad cl.name = c.name + ``\_Id" \; and \; cl.type = `NUMBER' \; and \\ \qquad\quad cl : k.column \; and \; cl.kind = `PrimaryKey' \; and \\ \qquad\quad cl : k.owner.column \end{array}$$

As with the tree to graph specification, these constraints define the expected state of the target model at completion of the transformation, with respect to the (unmodified) source model.

They can also be interpreted as describing individual transformation steps, and can be used to derive operation definitions. For example, $C2$ leads to the following operation on *Class*:

$$\begin{array}{l} \mathit{mapToTable}() \\ \texttt{post:}\ \mathit{kind} = \text{`}\mathit{Persistent}' \ \ \mathit{implies} \\ \quad \mathit{Table}{\rightarrow}\mathit{exists}1(\ t \mid t.\mathit{rdbId} = \mathit{umlId} \ \ \mathit{and} \ \ t.\mathit{name} = \mathit{name} \ \ \mathit{and} \\ \qquad t.\mathit{kind} = \text{`}\mathit{Persistent}' \ \ \mathit{and} \\ \qquad \mathit{Column}[\mathit{attribute}.\mathit{umlId}] \subseteq t.\mathit{column}\ ) \end{array}$$

The operations are iterated in successive phases, so that the target model is constructed from basic elements (columns), then tables are constructed from columns, primary and foreign keys depend on tables, and schemas are constructed from tables. The primary keys *umlId* and *rdbId* play the role of the explicit trace in VIATRA, by linking the source elements to the target elements that are derived from them.

6.5 Kermeta

The Kermeta solution follows a phased strategy at the top level similar to that of VIATRA. Classes are mapped to tables, and then attributes mapped to columns, and associations mapped to foreign keys. In this version also, the tracing facility is used to store the pairs of classes and their derived tables, and then to look up the tables of classes. Recursion is used to process attributes of class type and inheritance.

The transformation class is:

```
class Class2RDBMS { /** Transformation trace: */
  reference class2table : Trace<Class, Table>
  reference fkeys : Collection<FKey>

  operation transform(inputModel : ClassModel) : RDBMSModel
  is do
    class2table := Trace<Class, Table>.new
    class2table.create
    result := RDBMSModel.new

    getAllClasses(inputModel).select{ c | c.is_persistent }.each{
      c | var table : Table init Table.new
           table.name = c.name
           class2table.storeTrace(c,table)
           result.table.add(table)
    }

   getAllClasses(inputModel).select{ c | c.is_persistent }.each{
     c | createColumns(class2table.getTargetElem(c), c, "")
   }

  fkeys.each{ k | k.createFKeyColumns }
end
```

The operation to map attributes to columns is:

```
operation createColumnsForAttribute(table: Table, att: Attribute,
prefix: String) is do
  if PrimitiveDataType.isInstance(att.type)
  then
    var c: Column init Column.new
    c.name := prefix + att.name
    c.type := att.type.name
    table.cols.add(c)
    if att.is_primary then table.pkey.add(c) end
  else
    var type : Class type ?= att.type
    if isPersistentClass(type)
    then
      var fk : FKey init FKey.new
      fk.prefix := prefix + att.name
      table.fkeys.add(fk)
      fk.references := class2table.getTargetElem(getPersistentClass(type))
      fkeys.add(fk)
    else
      createColumns(table, type, prefix + att.name)
    end
  end
end
```

Here also, tracing is used to enable a phased development of the transformation. The query $class2table.getTargetElem(cls)$ finds the table previously derived from cls, and associated to cls by $class2table.storeTrace(cls, table)$.

7 Case Study 3: Quality Improvement

This case study is a typical example of an update-in-place quality improvement transformation. Its aim is to remove from a class diagram all cases where there are two or more sibling or root classes which all own a common-named and typed attribute.

It is used as one of a general collection of transformations (such as the removal of redundant inheritance, or multiple inheritance) which aim to improve the quality of a specification or design level class diagram.

Figure 5 shows the metamodel for the source and target language of this transformation.

It can be assumed that:

- No two classes have the same name.
- No two types have the same name.
- The owned attributes of each class have distinct names within the class, and do not have common names with the attributes of any superclass.
- There is no multiple inheritance.

These properties Asm must also be preserved by the transformation.

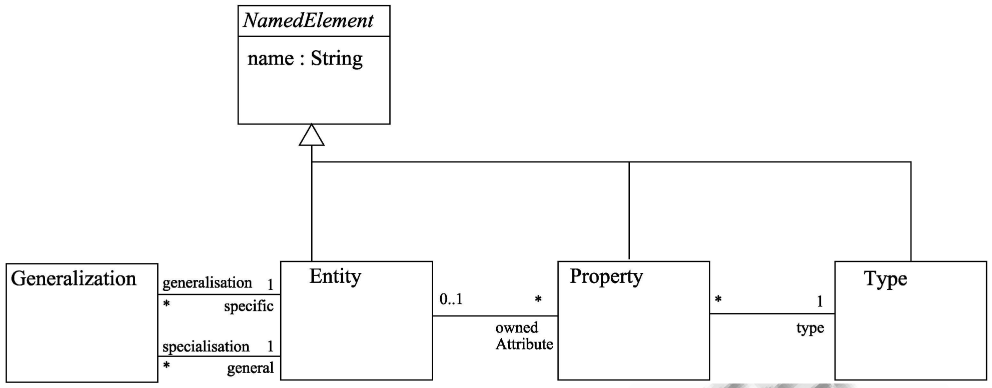


Figure 5. Basic class diagram metamodel

The informal transformation steps are the following:

- If a class c has two or more immediate subclasses $g = c.\mathit{specialization}.\mathit{specific}$, all of which have an owned attribute with the same name n and type t, add an attribute of this name and type to c and remove the copies from each element of g.
- If a class c has two or more immediate subclasses $g = c.\mathit{specialization}.\mathit{specific}$, and there is a subset $g1$ of g, of size at least 2, all the elements of which have an owned attribute with name n and of type t, but there are elements of g without such an attribute, introduce a new class $c1$ as a subclass of c and as a direct superclass of all those classes in g with the attribute. Add an attribute of this name and type to $c1$ and remove from each of its direct subclasses.
- If there are two or more root classes all of which have an owned attribute with name n and of type t, create a superclass c of all such classes and add an attribute of this name and type to c and remove from each of its direct subclasses.

It is required to minimise the number of new classes introduced, ie, to prioritise rule 1 over rules 2 or 3.

Unlike the previous examples, rules may be applied repeatedly to the same elements, and the application of a rule may affect subsequent rule applications, and so confluence and termination are non-trivial aspects of the problem. We will consider possible solutions in QVT-R, UML-RSDS and Kermeta.

7.1 QVT-relations

Rule 1 can be encoded as follows in QVT-R, using an adaption of the *PullUpAttribute* rule of Ref. [24]:

```
top relation PullUpDuplicateAttribute { primitive domain n : String;
  checkonly domain uml t : Type {};
  checkonly domain uml c : Class {};
  enforce domain uml p : Property { owner = c, name = n, type = t };

  when
  { c.specialization->size() > 1;
    c.specialization.specific->forAll(ownedAttribute->exists(name = n and type = t));
  }
}

top relation RemoveFromSubclasses { checkonly domain uml c : Class
  { specialization = g : Generalization { specific = sb : Class {} } };
  enforce domain uml p : Attribute { owner = sb };

  when
  { sb.ownedAttribute->includes(p);
    c.allAttribute().name->excludes(p.name);
  }
}
```

The first relation matches any class that has more than one subclass, all containing an attribute with the given name and type, and adds an attribute of this name and type to the class (unless one already exists). The second relation removes from all subclasses all attributes with a name that clashes with a superclass attribute name. It must be executed after the first relation.

The other rules can be specified in a similar way. The ability of QVT-R rules to match multiple elements (in this case, a class, name and type) simplifies the specification and makes it close in form to the requirements. On the other hand, there is no simple way to express the priority of one rule over another when both are enabled.

7.2 UML-RSDS

Rule 1 can be formalised as a constraint on *Class*:

$$\begin{array}{l}
\forall\, c : \mathit{Class} \;\cdot\; c.\mathit{specialization}\!\rightarrow\!\mathit{size}() > 1 \;\mathit{implies} \\
\quad \forall\, a : c.\mathit{specialization}.\mathit{specific}.\mathit{ownedAttribute} \;\cdot \\
\qquad c.\mathit{specialization}.\mathit{specific}\!\rightarrow\!\mathit{forAll}(\\
\qquad \mathit{ownedAttribute}\!\rightarrow\!\mathit{exists}(\mathit{name} = a.\mathit{name} \;\mathit{and}\; \mathit{type} = a.\mathit{type})) \;\mathit{implies} \\
\qquad\qquad \exists\, p : \mathit{Property} \cdot p.\mathit{name} = a.\mathit{name} \;\mathit{and}\; p.\mathit{type} = a.\mathit{type} \;\mathit{and} \\
\qquad\qquad\quad p : c.\mathit{ownedAttribute} \;\mathit{and} \\
c.\mathit{specialization}.\mathit{specific}.\mathit{ownedAttribute}\!\rightarrow\!\mathit{select}(\mathit{name} = a.\mathit{name})\!\rightarrow\!\mathit{isDeleted}()
\end{array}$$

The other two rules can be similarly formalised as constraints. In contrast to QVT-R, multiple matching is not directly supported, but is only simulated by means of multiple quantifiers.

From the constraint, design-level operations can be derived. Since the constraint both reads and modifies the same data (*ownedAttribute*), it requires a more complex implementation than the simple *for*-loop iteration of the previous examples. Instead, a schematic iteration of the form:

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

can be used. This can be explicitly coded as:

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

where:

```
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.

Priority of rule 1 over rules 2 and 3 is indicated by listing the constraint for rule 1 before the other constraints. In the implementation it is then attempted first in each iteration of the *search* for-loop.

7.3 Kermeta

The following is a part of the implementation of the first rule in Kermeta:

```
class QualityImprovement {
  reference quality_improvement : Trace<EMM1, EMM2>

  operation transform(inputModel : Root) : Root is do
    quality_improvement := Trace<EMM1, EMM2>.new
    quality_improvement.create
    getAllClasses(inputModel)
    result:=inputModel
  end


  operation getAllClasses (input:Root): EMM1[0..*] is do
    result := OrderedSet<EMM1>.new
    var myCol1 : set Property[0..*] init kermeta::standard::Set<Property>.new
        input.nas.each{c |
          if c.getMetaClass == EMM1 then
             myCol1 := getSharedProperty(c.asType(EMM1))
             updateModel(c.asType(EMM1),myCol1)
          end
        }
  end

operation updateModel(c1 :EMM1 , myCol1 : set  Property[0..*]) is do
    myCol1.each{p|stdio.writeln(p.name)
        c1.specialisation.each { e |
          e.specific.ownedAttribute.each{p1 |
               if p1.name == p.name and p1.type.name == p.type.name then
                 e.specific.ownedAttribute.remove(p1)
               end
           }
      }
      c1.ownedAttribute.add(p)
   }
   end
operation getSharedProperty(cl :EMM1 ) : set  Property[0..*] is do
        var myCol1 : set Property[0..*] init kermeta::standard::Set<Property>.new
      var myCol2 : set Property[0..*] init kermeta::standard::Set<Property>.new
      if  cl.specialisation.size() > 1  then // if it has any children
       myCol1 := getAllProperty(cl.specialisation.elementAt(0).specific)
       cl.specialisation.each { e |
         myCol2 := getAllProperty(e.specific)
         myCol1.each{ p | if not
          myCol2.exists{p1|p1.name == p.name and p1.type.name == p.type.name} then
               myCol1.remove(p)
            end
            }
         }
      }
      end
      result:= myCol1
      end

operation getAllProperty (input:EMM1): set Property[0..*] is do
    result := kermeta::standard::Set<Property>.new
    input.ownedAttribute.each{prs | result.add(prs) }
end
```

This solution finds all attributes that are common to all subclasses of a given class (*getSharedProperty*), then removes these common properties from the subclasses (*updateModel*). The algorithmic complexity is high, involving multiple nested iterations, although the program logic closely follows the specification.

8 Comparison

In this section we consider the evaluation criteria of Section 4 for each transformation specification notation and case study.

8.1 QVT-relations

The QVT-R notation is at a relatively high level of abstraction, close to the form of requirements, so in principle specifications in the notation can be directly validated. Each QVT-R rule has a mathematical interpretation as a predicate, so permitting, in principle, formal verification.

The modularity of the notation is only at the level of individual rules, which can however be interdependent and even mutually recursive. Since the notation is based closely upon UML and OCL, and the use of MOF metamodels, interoperability with other UML-based development tools is facilitated. OCL tools could be used to check the syntax and type-correctness of the rule patterns. However the implicit ordering of rule applications has the consequence that the transformation computed by a collection of rules may be unclear to the transformation developer. Syntactic correctness and definedness for QVT-Relations specifications can be analysed by expressing these properties as logical formulae in OCL and then using an analysis tool for OCL[3]. Check-before-enforce semantics in QVT means that if a result matching can be achieved using existing elements, then new elements are not created. This has a similar effect to an $\exists_1$ quantifier, and means that the creation of duplicate edges in the tree to graph case study is avoided. The facility to match multiple elements as inputs to a rule is useful, especially for re-expression and quality-improvement transformations.

Confluence of transformations may be difficult to establish, because the execution order of rules is only implicitly expressed in the rules. In the tree to graph example rule applications $Tree2Node(t)$ and $Tree2Node(t.parent)$ must be completed before a rule application $Tree2Edge(t)$ is completed, because of the *where* clause in the $Tree2Edge$ rule, however this leaves open many alternative orders of execution. Likewise in the other case studies. Proof of confluence is therefore necessary, to ensure that alternative permitted orderings of rule applications do not produce different target models. Termination may fail even for apparently trivial specifications[24]. Trace links are automatically created and managed in QVT-R and do not need to be defined by the developer. Transformations are executed by an interpreter.

8.2 ATL

ATL rules are at a relatively high level of abstraction, and a logical interpretation of the rules, similar to that of QVT-Relations rules, can be constructed, to aid comparison with requirement constraints. OCL tools could be used to check the syntax and type-correctness of the rule patterns. Modularity is at the level of rules, which may be interdependent both implicitly and explicitly (if one rule explicitly invokes another). ATL is interoperable with other MOF-based development tools[10]. The implicit execution order of ATL rules may hinder usability, as with QVT-Relations. There is no direct tool support for syntactic correctness or definedness, although the techniques of Ref. [3] could in principle be used to support this. There are no checks for rule completeness or semantic correctness. There are checks for rule consistency: if two rules are both applicable at the same time to the same model, an error is given. This helps to identify some cases of ambiguity in the specification, but does not ensure confluence in all cases.

There is no direct support for multiple input element matching, or for update-in-place transformations, which may hinder its applicability for quality improvement transformations. Trace links are automatically created and managed and do not need to be defined by the developer. ATL rules are compiled into an executable format, ATL VM[10].

8.3 VIATRA

The declarative style of VIATRA specification is at a relatively high level of abstraction, and VIATRA specifications are organised in a modular manner, as modules with sets of rules and a control algorithm. In addition, VIATRA allows the factoring out of parts of rules into separate procedures, which can then be reused in several rules. Such a mechanism is particularly useful in transformations involving complex expressions, such as the third case study above.

To use VIATRA with other development methods, input and output converters need to be specified. Usability may be hindered by the unusual notation and formalism used. Syntactic correctness is not checked, nor is definedness or completeness or semantic correctness. Confluence problems may arise, because some ASM constructs, such as *forall*, execute their statements in an indeterminate order. Matching of multiple input elements by rules is directly supported, as are update-in-place transformations. The hybrid nature of the notation gives a high degree of flexibility, enabling the specification of transformations using a combination of declarative and imperative styles.

Tracing needs to be managed explicitly by the developer. The rules can be executed by interpretation, or compiled to a directly executable form. The expression language used in VIATRA is simpler than OCL, and so may be more effectively verified.

8.4 UML-RSDS

The level of abstraction of the constraint-based specification of UML-RSDS is relatively high, and is close in form to requirements, without additional syntax, so facilitating direct validation. Modularity is based upon the object-oriented modularity of UML models (class diagrams and behaviour models). Transformations are characterised as use cases consisting of sets of constraints. These use cases may be externally composed by sequencing and conditional choice, or invoked as sub-procedures (the *includes* composition of use cases), or combined using conjunction (the *extends* composition of use cases) subject to their correctness constraints (assumptions *Asm* and effects *Cons*). Complex sub-expressions within constraints can be factored-out as query operations. Since UML-RSDS is based upon MOF and UML, interoperability with other UML development methods and tools should be possible. Usability should be high, since no new notation beyond UML is required.

The correctness of the generated operations with respect to the constraints is ensured by reasoning in the axiomatic semantics (Chapter 6 of Ref. [19]) of UML-RSDS. For example, in the first case study we can show that

$$t : Tree \;\Rightarrow\; [t.mapTreeToNode()](\exists\, n : Node \cdot n.name = t.name)$$

and hence

$$[for\ \ t : Tree\ \ do\ \ t.mapTreeToNode()](\forall\, t : Tree \cdot \exists\, n : Node \cdot n.name = t.name)$$

because the individual applications of *mapTreeToNode*() are independent and non-interfering. $[stat]P$ is the weakest precondition of predicate P with respect to statement *stat* (Chapter 6 of Ref. [19]).

Syntactic correctness of individual rules can be proved by using an automated translation from UML-RSDS to the B formal notation[16], and applying proof within B[20]. Definedness is checked by the generation of syntactic conditions (e.g., that the precondition of each called operation is true at the point of call) for the transformation design.

The UML-RSDS tools check completeness of rules by checking that all data features of an object are set in the operation which creates it. For example, in the operation *mapTreeToEdge*, an error message would be given if there was no assignment to the *target* of the new edge. Procedures for checking general semantic correctness exist but have not been automated. There are rules to determine when unordered iterations are confluent[20], which are implemented by checks on the write and read frames of constraints: confluence holds if these frames are disjoint for each constraint, and if the written data is updated in an order-independent manner. This condition holds in the first two case studies, but not in the third, so requiring a more complex implementation strategy.

Matching of multiple input elements is not directly supported, but update-in-place transformations are supported. A combination of declarative and imperative specification styles can be used.

Tracing of rule applications must be performed explicitly by the developer if required. The use of identity attributes is recommended as an alternative means of propagating changes from the source model to the target model, by identifying which target model elements are semantically linked to which source model elements (in the first case study, a tree is implicitly linked to the graph node with the same name, likewise for tables and classes in the second case study, via *umlId* and *rdbId*), independently of which model transformation rules were used to create the target model from the source. Implementation is by translation to Java. The generated code is close in structure to the specification, in the cases of refinement and re-expression transformations, which assists in comprehension.

8.5 Kermeta

The level of abstraction of Kermeta specifications is relatively low compared to the other approaches, the specification explicitly defines model management steps (such as the addition of new elements to the target model) which are implicit in the other notations considered here. Kermeta allows the definition of pre and post conditions for operations (and invariants for classes). These can be checked at runtime to detect erroneous processing of models, and represent a step towards hybrid specification. Modularity is provided in an object-oriented manner, by classes and packages. Transformations may be easily combined using programming constructions. Kermeta provides mechanisms to import and export models as objects. Interoperability with other MOF-based tools is possible. In contrast to QVT-Relations and ATL, the order of rule applications is explicit, and must be defined by the programmer. While this can result in larger and more complex specifications, the direct control by the developer over the transformation processing may reduce the possibilities of semantic errors and reduce development effort.

Matching of multiple input elements must be programmed explicitly by searches through models, as in the third case study example. Update-in-place transformations are supported.

There is no direct support for establishing syntactic correctness, however assertions and invariants could be used to check this for individual models. There is no support for checking definedness, completeness, semantic correctness or confluence. Tracing must be implemented explicitly by the developer. The language is itself executable.

8.6 Summary

Table 1 summarises the differences between the approaches we have surveyed. Under 'Specification' we consider the level of abstraction of the approach, whether the order of application of transformation rules are given explictly or implicitly, and what form of specification structuring and modularity is provided.

Table 1 General properties of model transformation approaches

Approach	*Specification*	*Validation*	*Verification*	*Implementation*
QVT-R	Abstract, implicit, rules	Tool checks on syntax	By OCL analyser	By interpretation
ATL	Abstract, implicit, rules	Tool checks on syntax, consistency	By OCL analyser	ATL virtual machine
VIATRA	Abstract, explicit, modules	Tool checks on syntax		Interpreter and compiler
UML-RSDS	Abstract, explicit, classes, use cases	Tool checks on syntax, confluence, completeness	B translation, inference rules	Code generation (Java)
Kermeta	Imperative, explicit, classes	Tool checks on syntax, testing	Embedded assertions	Already executable

Under 'Validation' we consider how a transformation specification in the approach can be validated against the transformation requirements, and checked for local correctness properties such as the consistency and completeness of individual rules. Under 'Verification' we consider how the specification can be checked for semantic correctness properties, such as confluence. Under 'Implementation' we consider how the transformation is executed.

With regard to the properties listed in Section 4, we have the following comparisons:

Level of abstraction Apart from the constraint-based specification approach in UML-RSDS, ATL is the notation at the highest level of abstraction, using implicit invocation of rules to compose transformation relations operating on composite elements and their components. QVT-R is also highly declarative, whilst VIATRA and UML-RSDS are hybrid and Kermeta is implementation-oriented.

Modularity The notations all support a concept of module, consisting of collections of closely-related rules. In ATL rules can be linked by implicit or explicit invocation, or by inheritance. In QVT-R there is implicit (for top relations) and explicit invocation, and VIATRA has explicit invocation. In UML-RSDS and Kermeta the usual object-oriented concepts of classes and objects are used to encapsulate groups of rules, as operations. Use cases are used to structure transformations in UML-RSDS. Explicit algorithms can be defined to control the order of rule applications within a module, in Kermeta, VIATRA and UML-RSDS. The usual composition of rules in UML-RSDS is a sequential phasing within a use case, which can also be used in VIATRA and Kermeta. VIATRA is the only notation with specific support for intra-rule modularisation, although this can be imitated by the use of query operations in other notations.

Interoperability VIATRA provides specific import and export facilities, and ATL, QVT-R (Medini QVT) and Kermeta can be executed within the Eclipse tool, supporting, in principle, interoperation with other Eclipse-hosted tools. UML-RSDS uses simple text file representations of models for input and output.

Usability In our case studies we found that developer effort was lowest in the case of Kermeta, because of the familiarity of the object-oriented programming paradigm, this required the least work to construct and analyse the specification. Implicit ordering of rule invocation, as found in ATL and QVT-R, particularly hindered comprehension and analysis. The unusual logic-programming paradigm of VIATRA also increased developer effort.

Syntactic and semantic correctness All the tools surveyed provide basic syntax checking of specifications, to identify errors in syntax, however there is a lack of semantic correctness analysis. UML-RSDS provides a completeness check on objects created by rules, and confluence checks for rules. ATL provides a runtime consistency check to detect situations where more than one rule is applicable to a model at one time. Other semantic checks, such as the detection of potentially unbounded recursion between rules, would also be beneficial for developers. Proof that metamodel constraints are established or preserved by transformations (syntactic and semantic correctness) is important in maintaining the integrity and correctness of a system, however only partial support is provided for syntactic correctness, and none for semantic correctness.

Definedness In UML-RSDS this is checked by syntactic conditions on the specification, that preconditions of rules are true at the point of their call, that there are no undefined expression evaluations in rules, and that loops terminate. Definedness can be checked for QVT-R and ATL by translation into OCL[3].

Rule completeness In UML-RSDS this is checked by examining the syntactic form of the rule postconditions. If an object is created in a postcondition, then all its data features should be assigned values in the postcondition, either explicitly or by implicit derivation.

Confluence ATL provides a check that no two rules are both enabled simultaneously on the same model, however this does not ensure confluence in all cases (a specification could contain a single rule which is iterated over a set of elements of the source model, with the order of execution being recorded in the target model). UML-RSDS provides syntactic restrictions to ensure confluence in some cases[20].

Change propagation Tracing facilities are provided in ATL, QVT, Kermeta and VIATRA. There is no direct support for change propagation in any of the approaches. QVT-R supports bidirectional transformations.

We compare the complexity of the approaches by applying metrics of size and call graph complexity to the specifications of the first and second case studies.

When comparing the specification sizes (measured by the lines of specification text, not including metamodel definitions), it should be considered that the VIATRA and Kermeta specifications both include model management code (as in the main method in Kermeta) that should be counted separately from the core transformation rules. In addition, the ATL and VIATRA specifications of case study two are somewhat simpler in scope than the others because they do not consider inheritance in the source model.

Table 2 shows the size metrics for the first two case studies. One surprising feature of these measures is that the declarative approaches of QVT and VIATRA have sizes larger than the imperative Kermeta approach, even though they omit explicit algorithmic details (completely so in the case of QVT). Both QVT and VIATRA include significant amounts of additional declaration syntax to establish the context of a transformation fragment, which are absent in the program-like syntax of Kermeta.

Table 2 Size metrics of case studies

Approach	*case study 1*	*case study 2*	*case study 3 (rule 1)*
QVT	27	180	19
VIATRA	22	180	
Kermeta	15	174	52
ATL	12	86	
UML-RSDS	11	75	7

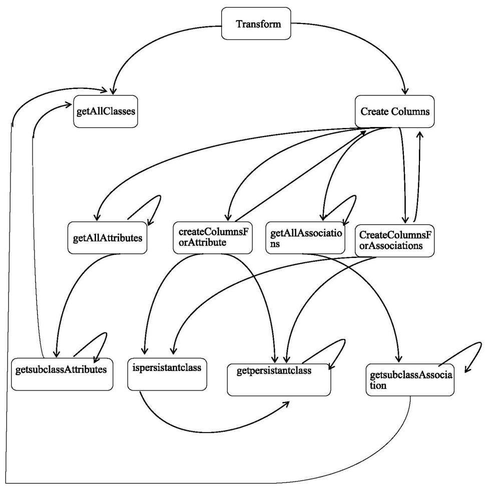
The complexity of a specification can be measured by analysing its call graph. A call graph is a directed graph that represents calling relationships between subroutines in a program. In the call graph each node represents functions or rules and edges represent calls between nodes. The size of the call graph affects the complexity of the program. The greater the number of arcs in the call graph, the higher is the dependency between different parts of the program, and so the greater is the complexity. Recursive calls have a substantial effect on complexity. Depending on the input model we can go through a recursion several times, which makes the transformation difficult to understand and verify.

Table 3 shows the call graphs of the different specifications on the second case study. Each node represents a rule and edges represent the calls from one rule to another, both implicit and explicit calls.

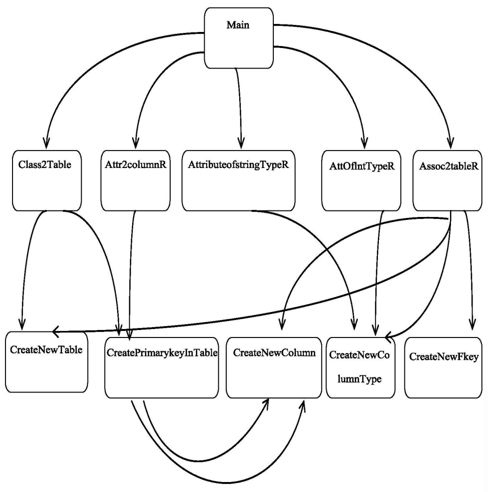
Table 4 compares the complexities of the approaches, based on the total number of calls and depth of calls. Depth1 is the maximum depth of call chains not involving recursive loops, and Depth2 the maximum length of a recursive loop.

This analysis shows considerable differences in the styles of specification adopted, with recursion being used substantially in some solutions (Kermeta and QVT), and not used at all in the other solutions. The problem does naturally lead to a recursive solution, because of the recursive structure of class diagrams (mapping a class to a relational table involves mapping its super or sub-classes also, and its owned attributes, which may be of a class type), however a phased solution is also possible, where basic elements (such as attributes of non-class type, and classes without subclasses) are mapped before elements composed from these elements. Such a phased solution could be defined in Kermeta. The QVT solution contains a potentially unbounded recursion (mapping attributes of a class type in the case of mutual dependencies between two classes), and only specifies local ordering restrictions between rules (for example, that the classes at either end of an association must be mapped before the association itself). This provides greater flexibility in execution order than a fixed scheduling of rules, and hence improves the potential for optimising execution efficiency, however it makes the proof of confluence and correctness of the transformation more difficult.

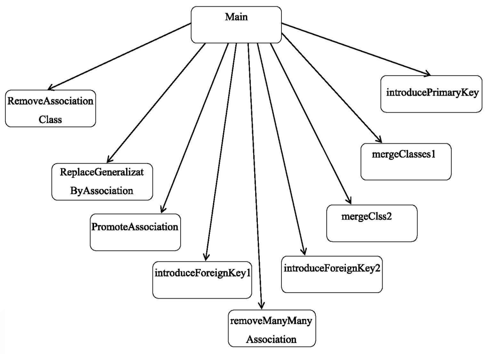
Table 3 Call graphs of UML to relational database schema transformations



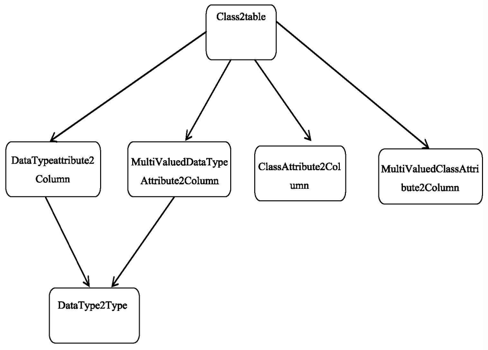
Kermeta



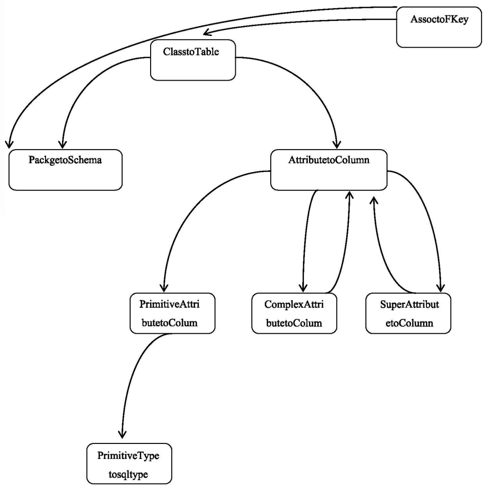
VIATRA



UML-RSDS



ATL



QVT

Table 4 Complexity of UML to relational database transformations

Approach	*Total number of calls*	*recursive calls*	*Depth2*	*non-recursive calls*	*Depth1*
Kermeta	22	9	2	13	4
VIATRA	16	0	0	16	3
QVT	10	4	2	6	4
UML-RSDS	11	0	0	11	1
ATL	6	0	0	6	2

In terms of the scope of the different approaches, all support multiple source model to multiple target model transformations. Only QVT-R directly supports bidirectional transformations, however. VIATRA has specific support for code generation (model to code transformations).

The structure of the source or target model often influences the modular decomposition of a transformation. Usually there is some hierarchical structure to the source language (eg, classes are parts of packages, and attributes parts of classes, in the UML 2 class diagram metamodel), and a model transformation can be organised according to this hierarchy: processing subordinate components of an element as part of the processing of that element. All the approaches described here support such hierarchical processing. An alternative is to use sequencing or phasing of processing, for example to order the transformation steps so that subordinate elements are mapped from source to target models before elements which refer to them (eg., in the tree to graph transformation in UML-RSDS and VIATRA, all tree nodes are mapped to graph nodes before creating edges that refer to the graph nodes). The approaches surveyed here do support phasing, although in QVT-R and ATL the hierarchical approach is more naturally expressed (as a recursive descent). Transformations may apply only to small numbers of elements within the model, leaving the remainder unchanged. In this case an efficient means for copying the unchanged source model elements to the target model is needed, or the transformation should be specified as an in-place update. ATL provides a particular mechanism (called the refining mode) to support such transformations. The other approaches described here support in-place update of models.

For refinement and re-expression transformations in particular, it is important that the transformation preserves semantic meaning. That is, the information of the source model is preserved in the target model, possibly under some interpretation. None of the approaches described here provides tool support to construct model interpretations and verify semantic correctness.

Table 5 compares the appropriateness of the approaches for the three different forms of transformation considered in this paper. For refinement and re-expression transformations either a phased or recursive descent strategy are usually applicable, with phased decomposition generally more modular and flexible. Quality-improvement transformations are usually update-in-place, requiring support for such a mechanism, and also have more complex behaviour than refinements or re-expressions, since the application of one restructuring step may affect subsequent steps: the transformation may be non-confluent and difficult to express except as local model rewrite rules. Matching of multiple elements in the source model for a single rule application may be necessary for re-expression and quality-improvement transformations, in contrast refinement transformations more usually map single source elements to multiple target elements.

Table 5 Appropriatenes of model transformation approaches

Approach	*Refinement*	*Re-expression*	*Quality improvement*
QVT-R	Recursive descent strategy, implicit phasing	Recursive descent strategy, implicit phasing	Rule-based specification usually concise and close to informal specification
ATL	Recursive descent strategy	Recursive descent strategy	Does not support update-in-place transformations
VIATRA	Phased or recursive strategies	Phased or recursive strategies	Multiple-element rule matching directly supported.
UML-RSDS	Phased strategy recommended	Phased strategy recommended	Complex specifications and code
Kermeta	Phased or recursive strategies	Phased or recursive strategies	Complex explicit algorithms

9 Conclusion

The significant differences between model transformation approaches concern their specification paradigm, levels of abstraction and choice of implicit or explicit rule sequencing. Our results suggest that the explicit programming paradigm of Kermeta is the most straightforward for novice model transformation developers to apply, at least on simple transformations. It may be expected that developers who are expert in UML would instead adopt more easily a notation close to UML, such as QVT-R or UML-RSDS. In general, explicit control over rule execution seems preferable for usability and analysis, however with the disadvantage of producing larger and more (apparantly) complex specifications. For refinement transformations relatively simple control strategies and matching strategies are usually sufficient, so favouring the use of more explicit approaches (Kermeta and the imperative features of VIATRA and UML-RSDS) with a phased implementation. For re-expressions, multiple input-element matching may be required in some cases (eg, where a group of source elements are amalgamated into a single element in the target), so requiring intrinsic support for such matching, as provided by VIATRA and QVT-R. If such matching is not required then the explicit approaches are recommended. For quality improvements, support for update-in-place transformation is necessary, as is support for multiple-element matching. Explicit approaches may involve extremely complex programming (as in the UML-RSDS and Kermeta solutions to case study three), so favouring the implicit style of QVT-R for such transformations.

The most important omission from many model transformation languages and methods is support for showing semantic correctness properties, both internal properties of a particular specification, such as definedness and confluence, and the effect of the transformation on the constraints of the source and target models.

Development approaches for model transformations have been formulated[7,13], however most model transformation development remains focussed upon the implementation level. Ideally, model-driven development should be applied to model transformations, with verification of the correctness and consistency of the transformations being carried out as an integral part of such development.

References

[1] Akehurst D, Howells W, McDonald-Maier K. Kent model transformation language. Model Transformations in Practice. 2005.

[2] Akehurst D, Kent S. A relational approach to defining transformations in a metamodel. Proc. UML 2002. LNCS 2460, Springer-Verlag, 2002.

[3] Cabot J, Clariso R, Guerra E, De Lara J. Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software, 2009 (preprint).

[4] Czarnecki K, Helsen S. Feature-based survey of model transformation approaches. IBM Systems Journal, 2006, 45(3): 621–645.

[5] Drey Z, Faucher C. et al.. Kermeta language reference manual. http://www.kermeta.org/docs /KerMeta-Manual.pdf. April 2009.

[6] Ehrig H, Engels G, Rozenberg HJ, *eds.* Handbook of graph grammars and computing by graph transformation. Volume 2, World Scientific Press, 1999.

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

[8] Jouault F, Kurtev I. Transforming Models with ATL. MoDELS 2005. LNCS 3844, Springer-Verlag, 2006. 128–138.

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

[10] Jouault F, Kurtev I. On the interoperability of model-to-model transformation languages. Science of Computer Programming, 2007, 68: 114–137.

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

[12] Kleppe A. 1st European workshop on composition of model transformations (CMT '06). Technical report TR-CTIT-06-34, University of Twente, 2006.

[13] Kolahdouz-Rahimi S, Lano K. A model-based development approach for model transformations. FSEN 2011. Iran.

[14] Kolovos D, Paige R, Polack F. The epsilon transformation language. ICMT 2008. LNCS 5063, Springer-Verlag, 2008. 46–60.

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

[16] Lano K. The B Language and Method. Springer-Verlag, 1996.

[17] Lano K. Constraint-driven development. Information and Software Technology, 2008, 50: 406–423.

[18] Lano K. A compositional semantics of UML-RSDS. SoSyM, February 2009, 8(1): 85–116.

[19] Lano K. (ed.).UML 2 Semantics and Applications. Wiley, 2009.

[20] Lano K, Kolahdouz-Rahimi S. Specification and Verification of Model Transformations using UML-RSDS. IFM 2010.

[21] Lano K, Kolahdouz-Rahimi S. Slicing of UML models using Model Transformations. MODELS 2010.

[22] Lano K, Kolahdouz-Rahimi S. Migration case study using UML-RSDS. TTC 2010. Malaga, Spain, July 2010.

[23] Lano K, Kolahdouz-Rahimi S. Model-driven development of model transformations. ICMT 2011. June 2011.

[24] Markovic S, Baar T. Refactoring OCL Annotated Class Diagrams. MoDELS 2005. LNCS 3713,

Springer-Verlag, 2005.

[25] Mens T, Czarnecki K, Van Gorp P. A taxonomy of model transformations. Dagstuhl Seminar Proc. 04101. 2005.

[26] OMG. UML superstructure, version 2.1.1. OMG document formal/2007-02-03. 2007.

[27] OMG. Query/View/Transformation Specification. 2009.

[28] OMG. Meta Object Facility (MOF) Core Specification. OMG document formal/06-01-01. 2006.

[29] OptXware. The VIATRA-I Model Transformation Framework Users Guide. 2010.

[30] Poernomo I. Proofs-as-Model-Transformations. Proc. of ICMT 2008. LNCS 5063, Springer-Verlag, 2008.

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

[32] Stevens P. Bidirectional model transformations in QVT. SoSyM, 2010, 9(1).

[33] Varro D, Pataricza A. Automated formal verification of model transformations. CSDUML 2003 Workshop. 2003.