Model Transformation Specification and Verification

Kevin Lano
Dept. of Computer Science, King's College London, Strand, WC2R 2LS
David Clark
Dept. of Computer Science, King's College London, Strand, WC2R 2LS

Abstract

Model transformations are becoming increasingly important in software development, particularly as part of model-driven development approaches (MDD). This paper defines an approach for specifying transformations as constraints, and for verifying the correctness of these transformations.

1. Introduction

The concepts of Model-driven Architecture (MDA) [10] and Model-driven Development (MDD) use model transformations as a central element, principally to transform high-level models (such as Platform-Independent Models, PIMs) towards more implementation-oriented models (Platform-specific Models, PSMs), but also to improve the quality of models at a particular level of abstraction. A standard for Queries, Views and Transformations (QVT) has been developed by the OMG [12], and defines different notations for specifying and implementing model transformations.

2. Categories of model transformation

Model transformations can be classified in six general categories:

Refinements These transformations are used to refine a model towards an implementation. For example, PIM to PSM transformations in the MDA. 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. Specialisation of a model at one level of abstraction could also be considered a special case of refinement in which the logical properties of models are strengthened: some situations which satisfy the original model will not satisfy the new model.

Quality improvements These transformations do not change the abstraction level or semantics of a model, but improve its structure and organisation, eg, by factoring out duplicated elements.

Elaborations These extend a model at the same level of abstraction by adding new elements, whilst not restricting the existing elements.

Re-expressions Translating a model in one language into its 'nearest equivalent' in a different language. This is useful for re-engineering, migration, validation and tool integration. Code generation, eg, from a UML Java PSM to Java, can also be considered to be in this category.

Abstractions The inverse of refinement transformations, these can be useful for reverse engineering, eg, from a PSM to a PIM, or for generalising a model.

Design patterns Reorganising a model to obtain a model that conforms to a standard pattern.

These categories overlap to some extent, for example, design patterns such as Template Method [7] can be regarded as quality improvement transformations.

3. A semantic theory of transformations

Transformations can be considered as being relations between models. The models may be in the same or in different modelling languages. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the languages concerned (in the case of UML these will typically be defined as metamodels which are subsets of the UML metamodels or variants of them). A transformation τ then describes which models M_1 of $\mathcal{L}_1$ correspond to (transform to) which models M_2 of $\mathcal{L}_2$.

1550-6002/08 $25.00 © 2008 IEEE
DOI 10.1109/QSIC.2008.38

τ can be expressed as a function or predicate defining the target element(s) of the transformation in terms of the source element(s). It may be that only some models in $\mathcal{L}_1$ can have τ validly applied to them: this is termed the *applicability condition* of τ.

An example, shown in Figure 1, is based on a re-expression transformation of [14]. On the left-hand side is a metamodel for the language $\mathcal{L}_1$, in which packages can be inherited by other packages, but associations are restricted to being unidirectional. On the right-hand side is a metamodel for the language $\mathcal{L}_2$, in which packages cannot be inherited by other packages, but associations can be unidirectional, bidirectional and undirected.

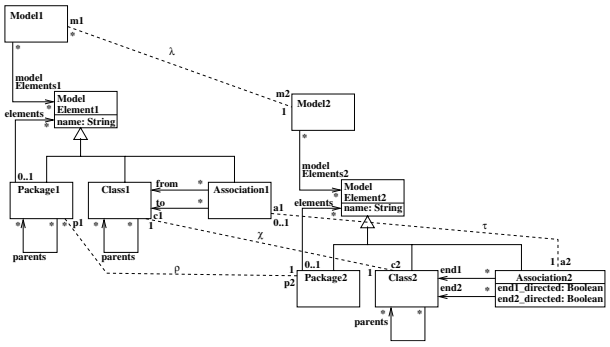

Figure 1. Transformation from $\mathcal{L}_1$ to $\mathcal{L}_2$

The transformation τ is expressed as a many-many (meta) association between the respective metaclasses $Association1$ and $Association2$ in the two languages. Given an $\mathcal{L}_1$ association $a1$, the corresponding $\mathcal{L}_2$ association $a2$ is defined by τ to have:

$$a2.name = a1.name$$
$$a2.end1\_directed = false$$
$$a2.end2\_directed = true$$
$$a2.end1 = a1.from.c2$$
$$a2.end2 = a1.to.c2$$

The transformation χ from $\mathcal{L}_1$ classes to $\mathcal{L}_2$ classes is essentially an identity relation. The union $\chi \cup \tau$ of these transformations can be applied to any $\mathcal{L}_1$ model which does not contain packages.

The QVT standard defines a graphical notation to describe transformations, using meta-object models. Figure 2 shows an example for the transformation τ. The left-hand side object model describes to which elements the transformation should be applied, the right-hand side shows the effect of the transformation. A *when* clause on the LHS can define additional applicability conditions, using OCL. A *where* clause defines further properties between the LHS and RHS which the transformation should establish, in this case that the association end classes are translated by $Class1\,ToClass2$ (χ).

C denotes that the LHS model is checked but not modified by the transformation, E denotes that the RHS model is modified if necessary to enforce the transformation relationship between the models.

This representation can be interpreted as a formal meta-association.

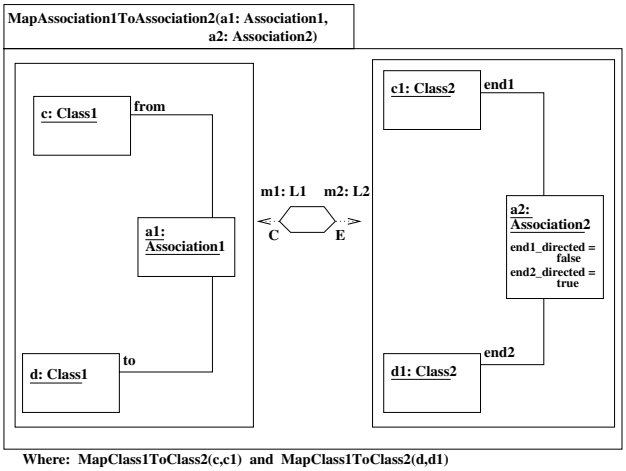

Figure 2. Transformation specification in QVT

Alternatively, sets of constraints can be used to specify the LHS and RHS, such as:

$$r1 : Association1$$

and

$$r2 : Association2$$
$$r2.name = r1.name$$
$$\chi(r1.from, r2.end1)$$
$$\chi(r1.to, r2.end2)$$
$$r2.end1\_directed = false$$
$$r2.end2\_directed = true$$

here.

Two important properties of a transformation are its *invertibility* and its *monotonicity*: a transformation is invertible if it can be applied in the reverse direction to give an identity relation on the source (language 1) elements/models, and monotonic if extensions of the source model transform into extensions of the target model.

The above example is invertible, but has restricted application on $\mathcal{L}_2$ because undirected and bidirectional associations in $\mathcal{L}_2$ have no representation in $\mathcal{L}_1$. It is monotonic.

We could also define a transformation ρ from $\mathcal{L}_1$ packages to $\mathcal{L}_2$ packages, which removes inheritance by amalgamating ancestor packages into descendent packages:

$$
\begin{array}{l}
(p1, p2) : \rho \ \Rightarrow \\
\quad p2.name = p1.name \ \wedge \\
\quad p2.elements \ = \\
\qquad (\tau \cup \chi \cup \rho)[p1.elements] \cup_P \\
\qquad \bigcup_P \rho[p1.parents]
\end{array}
$$

where $\cup_P$ represents package union.

This is many-one because differently-defined packages in $\mathcal{L}_1$ may flatten to the same package in $\mathcal{L}_2$. This transformation is therefore not invertible. In addition $\mathcal{L}_2$ packages containing undirected or bidirectional associations cannot be represented as $\mathcal{L}_1$ packages.

Finally, the union relation $\rho \cup \chi \cup \tau$ gives a transformation from any element of $\mathcal{L}_1$ to any element of $\mathcal{L}_2$.

3.1. Transformation specification by constraints

We will specify transformations between models by pairs L, R of constraints, L specifies what parts of the source (original) model are to be transformed, and R defines how they are transformed to produce the target (new) model.

L is effectively the precondition of an operation that transforms the model (ie, an operation at the metamodel level), and R is the postcondition. Expressions $e@pre$ in R refer to the value of e in the original model.

For example, consider a transformation to add primary keys to persistent entities without such keys. L could be:

$$
\begin{array}{l}
c : Class \\
\text{“}persistent\text{”} : c.stereotypeNames \\
\text{“}identity\text{”} \ / : \\
\quad c.ownedAttribute.stereotypeNames \\
c.name + \text{“}Id\text{”} \ / : \ c.feature.name
\end{array}
$$

using the metamodel of UML 2 [11], and where $m.stereotypeNames$ gives the set of names of stereotypes attached to a model element m.

L will be true for any model element c which is a persistent class without an identity attribute. For such elements the transformation defined by R will be applied to create a new model from the old, in which a new identity attribute is added to the selected class:

$$
\begin{array}{l}
a : Property \\
a.name = c.name + \text{“}Id\text{”} \\
a.stereotypeNames = \{\text{“}identity\text{”}\} \\
c.ownedAttribute \ = \\
\quad (c.ownedAttribute)@pre + Sequence\{a\} \\
a.classifier = c \\
a.type = IntegerType
\end{array}
$$

Because a occurs in R but not in L, it is assumed that it is created by the transformation, we could write $a.oclIsNew()$ to make this explicit.

The operation specified by this pre and post condition is named $introducePrimaryKey()$. It will be repeatedly applied to a model while classes which satisfy the precondition remain in the model.

3.2. Model transformation correctness

The correctness of a model transformation τ from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$ can be considered at syntactic and semantic levels:

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 $\mathcal{L}_2$.

Semantic correctness With respect to the semantics of $\mathcal{L}_1$ and $\mathcal{L}_2$ being used, if a model $M1$ of $\mathcal{L}_1$ is transformed to a model $M2$ of $\mathcal{L}_2$, then each property of $M1$ true under the $\mathcal{L}_1$ semantics is also true, under the interpretation ζ induced by τ, in $M2$ under the $\mathcal{L}_2$ semantics.

The second property should be expected for refinement, specialisation, enhancement, quality improvement and design pattern transformations. For abstractions it will instead be the case that all $M2$ properties should be valid in $M1$. For re-expression transformations there may be cases where $M1$ properties cannot be expressed in $M2$, but all expressable properties should be preserved from $M1$ to $M2$.

More precisely, semantic correctness means that the diagram of Figure 3 commutes: each formula $\varphi \in \Gamma_1$ has

$$\Gamma_2 \vdash \zeta(\varphi)$$

where Γ_1 is the semantics of M_1 under $Sem1$, and Γ_2 is the semantics of M_2 under $Sem2$.

For the remainder of this paper we will use the UML 2 metamodel or subsets as the model languages, and give a semantics to models by expressing them as sets

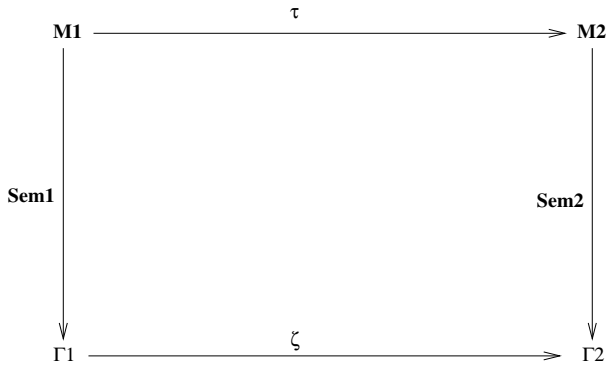

Figure 3. Transformation correctness

of constraints (in an OCL-like constraint language) in suitable theories.

An often neglected consideration is that not only the graphical elements of models change under a transformation, but also its constraints may need to change, in order to be correct interpretations in the new model of the constraint in the source model. For example, a constraint $from = to$ defining a self-association in $\mathcal{L}_1$ in Figure 1 would need to be transformed to $end1 = end2$ in $\mathcal{L}_2$.

In general, a constraint φ should be transformed to $\zeta(\varphi)$ or to a predicate which implies this.

It is often the case that groups of related transformations are used together to transform a model. For example, the transformations to form a relational database schema from a class diagram (introduce primary and foreign keys, remove inheritance, many-many associations and association classes), described in Section 5, would normally be used in this way. For such groups, the property of *consistency* is important: it should not be the case that two transformations in the group can both be applied to the same model and produce different results.

If a group fails to satisfy this property (eg, as is the case for the 'introduce primary keys' and 'amalgamate classes' transformations), then a definite order of application or priority scheme must be defined to remove such conflicts. In this case the ordering could be:

1. Eliminate inheritance;
2. Introduce primary keys;
3. Eliminate many-many associations and association classes (the primary keys of the classes at the association ends can be used together as a compound key of the intermediate class);
4. Implement many-one associations by foreign keys.

This defines an algorithm, which may be expressed as a behaviour state machine (Figure 4). The correctness of this algorithm can be shown by defining suitable state invariants and loop variants.

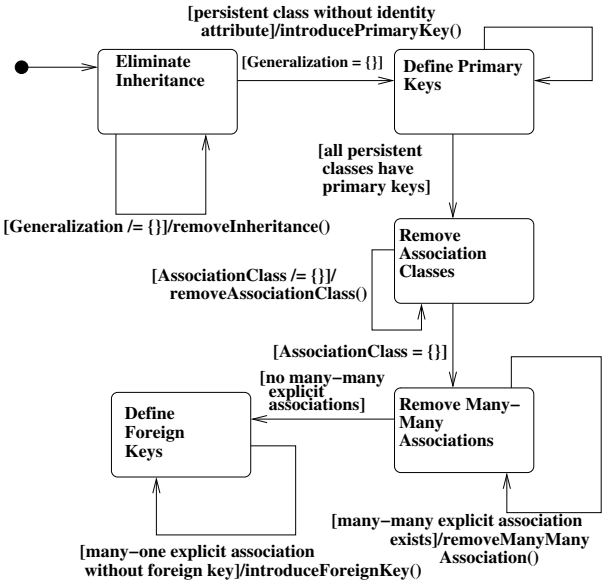

Figure 4. Transformation algorithm

For instance, the UML 2 class diagram property that all classes have no same-named features:

$$Class.allInstances() \rightarrow forall(\\ features \rightarrow size() = \\ features.name \rightarrow size())$$

is preserved by the algorithm, because each of the individual transformations preserve it.

4. Catalogue of model transformations

In the following sections, we provide examples of UML transformations and their formalisation. Each transformation is described, with a brief reference to its purpose, effect and provenance.

For design patterns, enhancements, specialisations and quality improvements, the transformations usually operate within a single language, ie, $\mathcal{L}_1 = \mathcal{L}_2$, in this case, the UML 2 metamodel. Syntactic correctness reduces to showing that the transformed model satisfies the restrictions of the metamodel if the original model does. For refinements, abstractions, and re-expressions, the source and target languages may be different.

5. Refinement transformations

Refinement transformations have the general aim to refine a model to a more implementation-oriented version. In the context of the MDA, this means transforming a computation-independent model (CIM) to a PIM, or a PIM to a PSM.

In moving from a PIM to a PSM, elements in the PIM which are not supported directly in the target platform must be removed from the model, and replaced by platform-specific elements which satisfy their semantics. For example, in refining a UML class diagram to a relational database data model, we must replace explicit many-many associations, association classes, qualified associations, and inheritance, and introduce primary and foreign keys [2]. In refining a PIM to a Java PSM, we must eliminate multiple inheritance and association classes.

Other forms of refinement transformation replace specification-level 'what' descriptions by design-level 'how' definitions. For example, by introducing a specialised algorithm.

5.1. Remove association classes

This refinement transformation removes an association class from a model. The matching predicate L in this case is

$$r : AssociationClass$$

Association classes are replaced by a class plus two new associations (Figure 5). The predicate R for the new model is, in part:

$$\begin{array}{l} c : Class \\ c.name = r.name \\ c.ownedAttribute = r.ownedAttribute \\ c.ownedOperation = r.ownedOperation \\ r\ /: \ Element \\ a1, a2 : Association \\ e11, e12, e21, e22 : Property \\ a1.memberEnd = Sequence\{e11, e12\} \\ a2.memberEnd = Sequence\{e21, e22\} \\ e11.type = r.memberEnd[1].type \\ e22.type = r.memberEnd[2].type \\ e12.type = c \\ e21.type = c \end{array}$$

This defines a new class c which has a copy of r's features, defines that r itself is removed from the model, and there are two new associations $a1$ and $a2$ which link c to the original end classes of r.

A new constraint

$$\begin{array}{l} r1 : A\_B \ \& \ r2 : A\_B \ \& \\ r1.a = r2.a \ \& \\ r1.b = r2.b \ \Rightarrow \ r1 = r2 \end{array}$$

holds in the resulting model.

The original role ar is implemented by the composition $ar".a$. Likewise br is implemented by $br".b$.

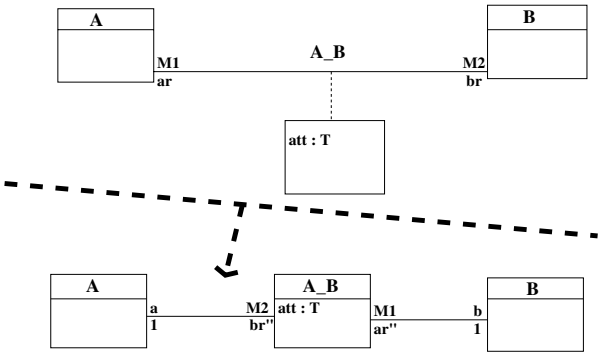

Figure 5. Transformation of association classes to associations

Syntactic correctness of the transformation is clear. The multiplicities $M1$ and $M2$ can be any multiplicities allowed in the modelling language. The uniqueness property defined above ensures that the interpretation of the multiplicity properties hold in the new model, eg., $br".b$ satisfies $M2$ if br does. Semantic correctness therefore follows.

This transformation is T20 in [3].

5.2. Amalgamate subclasses into superclass

This refinement transformation amalgamates all features of all subclasses of a class C into C itself, together with an additional flag attribute to indicate which class the current object really belongs to. It is one strategy for representing a class hierarchy in a relational database.

Its matching condition L is

$$\begin{array}{l} c : Class \ \& \\ c.generalization = \{\} \ \& \\ c.specialization \ /= \ \{\} \end{array}$$

An example of the transformation is shown in Figure 6.

To ensure semantic correctness, constraints of the subclasses must be re-expressed as constraints of the amalgamated class, using the flag attribute, as illustrated in Figure 6.

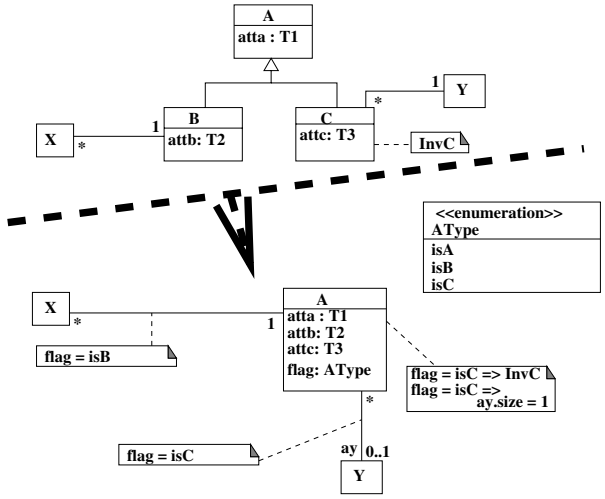

Figure 6. Amalgamation of subclasses transformation

The transformation can be applied in a series of steps, which each move one direct subclass d of c up to c, together with the associations connected to d. For each removed subclass, a new element is added to the enumerated type for the flag attribute.

This transformation is related to the Collapsing Hierarchy refactoring of [6].

Other class diagram refactorings based upon those of [6] are given for UML in [9]: Rename (class, attribute, role, operation); PullUp (attribute, role, operation); PushDown (attribute, role, operation); Extract (class, superclass); Move (attribute, operation, role).

5.3. Replace inheritance by association

This refinement transformation is an alternative way of removing inheritance, it replaces an inheritance relationship between two classes by an association between the classes.

The matching condition is:

$c1 : Class$ &
$c2 : Class$ &
$c1 : c2.generalization.general$

In contrast to the amalgamation approach it allows a subclass and a superclass to be represented in different database tables. This is useful if the classes will be processed in different ways (eg, in different use cases) in the application, or if amalgamating them would produce tables with an excessive number of columns.

As a meta-association "replace inheritance by association" relates an element $g : Generalization$ to an $a : Association$ and $r1, r2 : Property$ such that

$$a.memberEnd = Sequence\{r1, r2\}$$
$$r1.classifier = g.general$$
$$r2.classifier = g.specific$$

and $r1$ has 0..1 multiplicity and $r2$ has 1 multiplicity. The transformation is invertible and monotone. However, not all 0..1 to 1 associations can be validly converted into inheritances: the classes at the ends of the association must be different, and the class at the 1 end must conceptually represent a generalisation of the other class. If both classes have primary keys, these should be the same.

Figure 7 shows the application of this transformation. The inheritance of B on A is replaced by a 0..1 to 1 association from B to A.

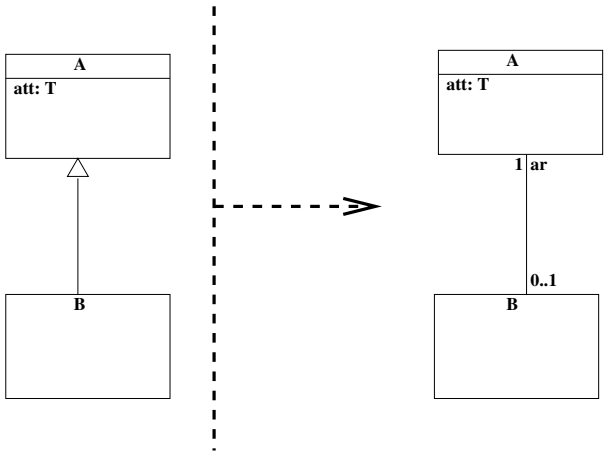

Figure 7. Replace inheritance by association application

To ensure semantic correctness, any expression in the original model which has B as contextual classifier, and which uses a feature f inherited from A, must be modified in the new model to use $ar.f$ instead.

The transformation is also used in [7] to improve the quality of models where inheritance would be misapplied, such as situations of dynamic and multiple roles. It is related to the Role pattern of [1].

5.4. Remove many-many associations

This refinement transformation replaces a many-many association with a new class and two many-one associations.

The L constraint is:

$r : Association$ &
"$explicit$" : $r.stereotypeNames$ &
$r.memberEnd[1].upper > 1$ &
$r.memberEnd[2].upper > 1$

Explicit many-many associations cannot be directly implemented using foreign keys in a relational database – an intermediary table would need to be used instead. This transformation is the object-oriented equivalent of introducing such a table.

The transformation is shown in Figure 8.

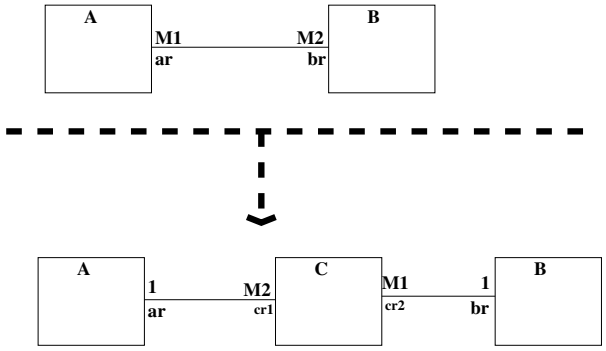

Figure 8. Removing a many-many association

To ensure semantic correctness, the new class must link exactly those objects that were connected by the original association, and must not duplicate such links:

$c1 : C$ & $c2 : C$ &
$c1.ar = c2.ar$ &
$c1.br = c2.br \Rightarrow c1 = c2$

In addition, any constraint with contextual classifier A or a subclass of A, which refers to br, must replace this reference by $cr1.br$ in the new model. Likewise for navigations from B to A.

5.5. Introduce primary key

This refinement transformation applies to any persistent class. If the class does not already have a primary key, it introduces a new identity attribute, usually of Integer or String type, for this class, together with extensions of the constructor of the class, and a new *get* method, to allow initialisation and read access of this attribute.

This is an essential step for implementation of a data model in a relational database. The transformation is shown in Figure 9.

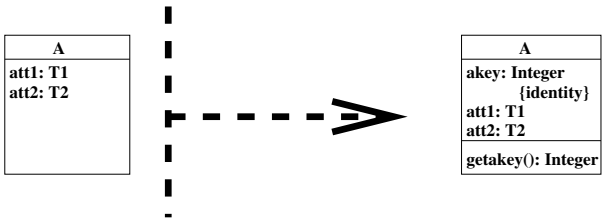

Figure 9. Introduce primary key

A new constraint expressing the primary key property is added to the new model:

$A.allInstances() \rightarrow size() =$
$A.allInstances() \rightarrow$
$collect(akey) \rightarrow size()$

This must be maintained by the constructor, for example:

```
A(att1x : T1, att2x: T2, akeyx: Integer)
pre:  akeyx /: A.allInstances->collect(akey)
post: akey = akeyx &
      att1 = att1x & att2 = att2x
```

5.6. Replace association by foreign key

This refinement transformation applies to any explicit many-one association between persistent classes. It assumes that primary keys already exist for the classes linked by the association. It replaces the association by embedding values of the key of the entity at the 'one' end of the association into the entity at the 'many' end.

The L constraint is:

$r : Association$ &
"$explicit$" : $r.stereotypeNames$ &
$r.memberEnd[1].upper = 1$ &
$r.memberEnd[2].upper > 1$

(together with the case that the member ends are in the opposite order).

This is an essential step for implementation of a data model in a relational database. The transformation is shown in Figure 10.

$b.akey$ is equal to $a.akey$ exactly when $a \mapsto b$ is in the original association. This correspondence must be maintained by implementing *addbr* and *removebr* operations in terms of the foreign key values.

Navigation from an A instance to its associated br set must be replaced by

$B.allInstances() \rightarrow$
$select(B :: akey = A :: akey)$

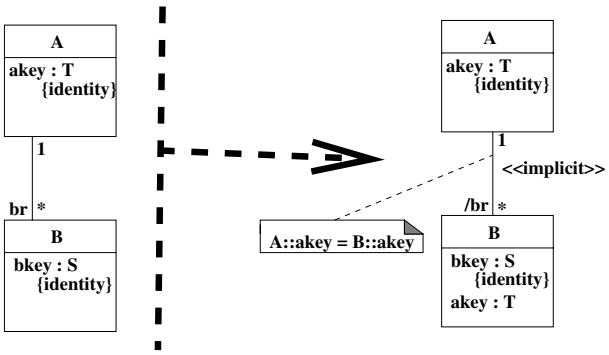

Figure 10. Replacing association by foreign key

in the new model, corresponding to an SQL *SELECT* statement. Likewise for navigation from B to A.

5.7. Replace global constraint by local constraints

This transformation refines a class diagram by replacing a constraint which spans n classes, $n > 1$, by constraints which are local to m classes, $m < n$. Local constraints are usually easier to implement than global constraints.

L in this case is:

$$c : Constraint \ \& \\ c.constrainedElements \rightarrow size() > 1$$

A typical form of localisation is when some global constraint can be replaced by one or more (more local) constraints, which together ensure the global constraint (Figure 11).

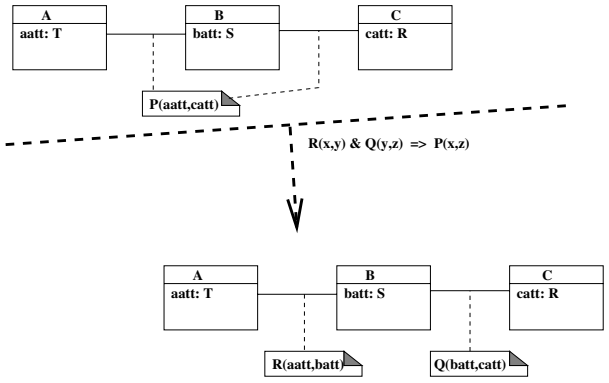

Figure 11. Localising constraints

For example, $P(aatt, catt)$ could be $aatt < catt$ and $Q(aatt, batt)$ is $aatt < batt$ and $R(batt, catt)$ is $batt \leq catt$.

6. Quality improvement transformations

These transformations do not change the abstraction level of a model, but rationalise and improve its structure and elements to make the model more flexible, concise, comprehensible or complete.

One subcategory is that of *factoring* and *refactoring* transformations: factoring transformations introduce a new element which captures commonalities between elements of the original model, which had no distinct representation in that model. Inheritance, in class diagrams, and state inclusion, in state machines, are typically used to carry out this factoring. Some design patterns, such as Template Method, Facade and Session Facade [5], can be also seen as factoring quality improvement transformations. Refactorings reorganise existing structure to improve the factorisation in the model, instead of introducing new elements [6].

A second subcategory is the removal of redundancy from a model, of elements which duplicate information which is already present in the model.

6.1. Introduce superclass

This quality improvement transformation introduces a superclass of several existing classes, to enable common features of these classes to be factored out and placed in a single location.

The L constraint could be:

$$c1 : Class \ \& \\ c2 : Class \ \& \\ (c1.feature.name \cap \\ c2.feature.name) \rightarrow size() > 1 \ \& \\ c1.generalization.general \cap \\ c2.generalization.general = \{\}$$

together with the condition that $c1$ is not a direct or indirect ancestor/descendent of $c2$.

In general, this transformation should be applied if there are several classes A, B, ... which have common features, and there is no existing common superclass of these classes. Likewise if there is some natural generalisation of these classes which is absent in the model.

It is particularly useful for reorganising and rationalising a class diagram after some change to a system specification [4].

Figure 12 shows a generic example, where the existing classes have both common attributes, operations and roles.

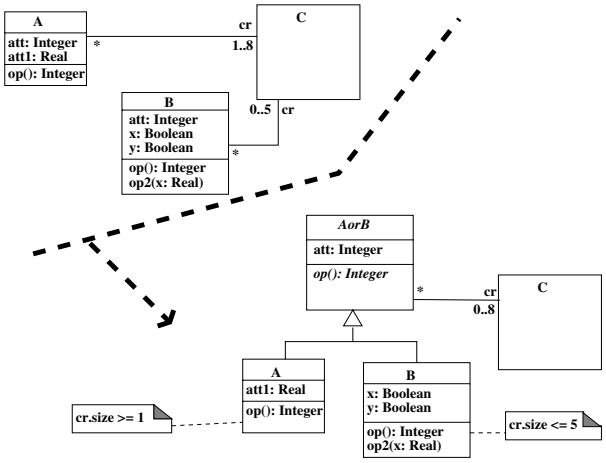

Figure 12. General superclass introduction

The features that are placed in the superclass must have the same intended meaning in the different subclasses, rather than an accidental coincidence of names.

The properties of the features in the superclass are the disjunction of their properties in the individual subclasses. For common roles, this means that their multiplicity on the association from the superclass is the 'strongest common generalisation' of their multiplicities on the subclass associations. Eg, if the subclass multiplicities were $m1..n1$ and $m2..n2$, the superclass multiplicity would be $min(m1, m2)..max(n1, n2)$. For common operations, the conjunction of the individual preconditions can be used as the superclass operation precondition, and the disjunction of the individual postconditions as the superclass operation postcondition.

Common constraints of the subclasses can also be placed on the superclass. Where a feature of a subclass is moved up to the superclass, its specific constraints as a feature of the subclass must be stated as constraints of the subclass, as is the case with *cr* in the above example.

Variations include situations where a common superclass already exists, but some common features of its subclasses are missing from it. In this case the common features are simply moved up to the superclass. The 'Pull up method' refactoring of [6] is one case of this situation.

6.2. Introduce superstate

If states s_1, ..., s_n of a state machine all have a common set of outgoing transitions, ie, for a non-empty set α_1, ..., α_m of events they have transitions t_{s_1,α_1}, ..., t_{s_n,α_1}, etc, such that, for a given j, the t_{s_i,α_j} all have the same guards, actions and target states, then introduce a new superstate s of the s_i, and replace the t_{s_i,α_j} by new transitions t_{s,α_j} from s to the common target of the t_{s_i,α_j}, and with the same guard and actions. Common invariants of the substates can be placed on the superstate.

This reduces the complexity of the diagram (the number of transitions is reduced by $(n-1)*m$) and may identify a conceptually significant state that was omitted from the original model.

Figure 13 shows an example of this transformation.

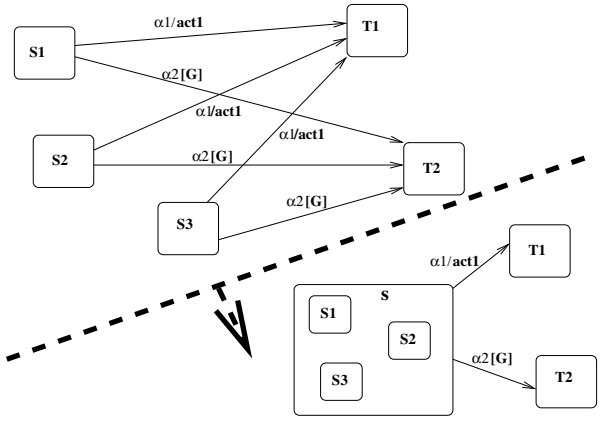

Figure 13. Introduce superstate

To ensure syntactic correctness, this transformation can only be applied if s only intersects its super- and sub-states.

7. Design patterns

Introducing a design pattern by re-organising the elements of a model to conform to the pattern can be regarded as a model transformation [8]. In some cases these will be quality improvement transformations, in other cases refinements [13]. The Abstract Factory, Adapter, State, Mediator and Observer patterns are analysed as transformations in [8].

8. Enhancement transformations

Enhancement transformations extend a model in a monotonic manner, such that all existing elements of the model remain in the new model, but new elements are added. The primitive transformations 'Add construct', 'Assert construct is derived' (together with the constraint which defines the derivation), 'Reorder attributes' of [3] are examples.

Not all additions to a diagram are valid enhancements. For example, adding a new element to an enumerated type falsifies the semantics of the original model, as does adding a new direct substate to a composite state with a single region. In both cases the extended element already had an axiom expressing that it was complete in the original model (although UML allows an alternative semantics for composite states which does not enforce completeness).

9. Summary

We have defined a systematic representation and classification of model transformations, and described examples of common transformations. The conditions necessary to ensure semantic correctness of these transformations have been described.

References

[1] D. Bämer, et al, *Role Object*, Pattern Languages of Program Design, Addison-Wesley, 2000.

[2] C. Batini, S. Ceri, S. Navathe, *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.

[3] M. Blaha, W. Premerlani, *A catalog of object model transformations*, 3rd Working Conference on Reverse Engineering, California, 1996.

[4] G. Booch, M. Engel, and B. Young, *Object Oriented Analysis and Design with Applications*, Addison-Wesley, 2007.

[5] J Crupi, D Alur, and D Malks. *Core J2EE Patterns*. Prentice Hall, 2001.

[6] M. Fowler, *Refactoring: Improving the design of existing code*, Addison-Wesley, 2000.

[7] M. Grand, *Patterns in Java*, Wiley, 1998.

[8] K. Lano, *Formalising Design Patterns as Model Transformations*, Chapter VIII of *Design Pattern Formalisation Techniques*, T. Taibi (ed.), IGI Publishing, 2007.

[9] Slavisa Markovic and Thomas Baar, *Refactoring OCL Annotated UML Class Diagrams*, MODELS 2005 Proceedings, LNCS 3713, Springer-Verlag, 2005.

[10] OMG, Model-Driven Architecture, http://www.omg.org/mda/, 2007.

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

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

[13] G Sunyé, A Le Guennec, and J M Jézéquel. Design Patterns Application in UML. In *ECOOP 2000*, number 1850 in Lecture Notes in Computer Science, pages 44–62. Springer, 2000.

[14] L. Tratt, *Model transformations and tool integration*, SoSym, Vol. 4, No. 2, 2005.