Design Patterns for Model Transformations

Kevin Lano
Dept. of Informatics
King's College London
London, UK
Email: kevin.lano@kcl.ac.uk

Shekoufeh Kolahdouz-Rahimi
Dept. of Informatics
King's College London
London, UK
Email: shekoufeh.kolahdouzrahimi@kcl.ac.uk

***Abstract*—Model transformations are a central element of model-driven software development. This paper defines design patterns for the specification and implementation of model transformations. These patterns are commonly recurring structures and mechanisms which we have identified in many specific transformations. In this paper we show how they can be used together to support an overall development process for model transformations from high-level specifications to executable Java implementations.**

Keywords —- Design patterns; model transformations; UML.

I. INTRODUCTION

Design patterns for software development were introduced by Gamma et al [5]. Subsequently, many hundreds of patterns have been identified, including patterns for specialised forms of development such as enterprise information systems [3]. Patterns for model transformations were proposed by [1]. In this paper, we consider further patterns, based on a large number of case studies which we have carried out or analysed. These patterns are inter-related and can be used together to support the development of transformations from high-level specifications as sets of constraints, to executable implementations in Java. They have been incorporated into our transformation environment, UML-RSDS [10].

Section II describes related work, Section III defines a general development process for model transformations. Section IV describes specification patterns, Section V describes implementation patterns and Section VI gives conclusions.

II. RELATED WORK

General design patterns can be used for model transformations. For example, the Builder and Abstract Factory patterns are directly relevant to transformation implementation, in cases where complex platform-specific structures of elements must be constructed from semantic information in a platform-independent model, such as the synthesis of J2EE systems from UML specifications. The Visitor pattern can be used for model-to-text transformations [4]. The Model-view-controller pattern is relevant for change-propagating model transformations, where changes to the source model are propagated to the target (view).

Patterns specific to model transformations have been identified and used previously. In [2], specifications of the conjunctive-implicative form (Section IV) are derived from model transformation implementations in triple graph grammars and QVT, in order to analyse properties of the transformations, such as definedness and determinacy. This form of specification is therefore implicitly present in QVT and other transformation languages.

In [14], [15] the concept of the conjunctive-implicative form was introduced to support the automated derivation of transformation implementations from specifications written in a constructive type theory.

In [1], a transformation specification pattern is introduced, *Transformation parameters*, to represent the case where some auxiliary information is needed to configure a transformation. This could be considered as a special case of the auxiliary metamodel pattern (Section IV). An implementation pattern *Multiple matching* is also defined, to simulate rules with multiple element matching on their antecedent side, using single element matching. We also use this pattern, via the use of multiple $\forall$ quantifiers in specifications and multiple *for* loops at the design level to select groups of elements.

Our work extends previous work on model transformation patterns by combining patterns into an overall process for developing model transformation designs and implementations from their specifications. The patterns are an essential part of the UML-RSDS development process for model transformations.

III. DEVELOPMENT PROCESS FOR MODEL TRANSFORMATIONS

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

For a transformation τ from S to T, there are three separate predicates which characterise its global properties, and which need to be considered in its specification and design [10]:

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-165-6

1) *Asm* – assumptions, expressed in the union language $\mathcal{L}_{S\cup T}$ of the source and target metamodels, which can be assumed to be true before the transformation is applied. These may be assertions that the source model is syntactically correct, that the target model is empty, or more specialised assumptions necessary for τ to be well-defined. These are preconditions of the use case of the transformation.
2) *Ens* – properties, usually expressed in $\mathcal{L}_T$, which the transformation should ensure about the target model at termination of the transformation. These properties usually include the constraints of *T*, in order that syntactic correctness holds. For update-in-place transformations, where the source and target languages are the same, *Ens* may refer to the pre-state versions of model data.
3) *Cons* – constraints, expressed in $\mathcal{L}_{S\cup T}$, which define the transformation as a relationship between the elements of the source and target models, which should hold at termination of the transformation. Update-in-place transformations can be specified by using a syntactically distinct copy of the source language, for example by postfixing all its entity and feature names by *@pre*.
   Cons corresponds to the postconditions of the use case of the transformation.

We can express these predicates using OCL notation, this corresponds directly to a fully formal version in the axiomatic UML semantics of [8]. Together these predicates give a global and declarative definition of the transformation and its requirements, so that the correctness of a transformation may be analysed at the specification level, independently of how it is implemented.

The following should be provable:

$$Cons, \Gamma_S \vdash_{\mathcal{L}_{S\cup T}} Ens$$

where Γ_S is the semantic representation of the source language as a theory.

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

$$\Gamma_S \vdash_{\mathcal{L}_{S\cup T}} Asm \ \Rightarrow\ [activity]Cons$$

where *activity* is the algorithm of the transformation. Each statement form of the statement language (Chapter 6 of [8]) has a corresponding definition of $[\ ]$.

IV. Specification Patterns

In this section we describe characteristic patterns for the specifications of model transformations.

A. Conjunctive implicative form

Synopsis: To specify the effect of a transformation in a declarative manner, as a global pre/post predicate, consisting of a conjunction of constraints with a $\forall \Rightarrow \exists$ structure.

Forces: Useful whenever a platform-independent specification of a transformation is necessary. The conjunctive-implicative form can be used to analyse the semantics of a transformation, and also to construct an implementation.

The pattern typically applies when *S* and *T* are similar in structure, for example in the UML to relational database mapping of [13], [10], the source structure of *Package*, *Class*, *Attribute* corresponds to the target structure of *Schema*, *Table*, *Column*.

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

$$\forall s : S_i \cdot SCond_{i,j} \ implies\ \exists t : T_{i,j} \cdot LPost_{i,j} \ and \ GPost_{i,j}$$

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

Figure 1 shows a schematic structure of this pattern.

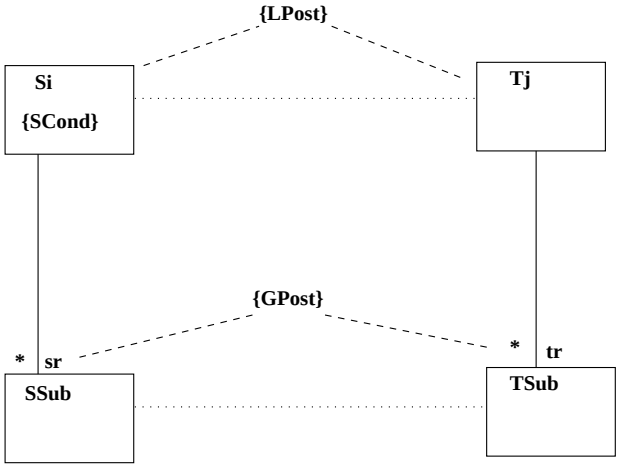

Figure 1. Conjunctive-implicative form

We distinguish three cases of constraints *Cn*:

1) Type 1 constraints: $rd(Cn) \cap wr(Cn) = \{\}$ where *rd* is the read frame and *wr* the write frame of the constraint: the set of features and entities which it (conceptually) reads and updates.
2) Type 2 constraints: $S_i \notin wr(Cn)$ and $rd(SCond) \cap wr(Cn) = \{\}$ but $rd(Cn) \cap wr(Cn) \neq \{\}$.
3) Type 3 constraints: all other cases.

For type 2 or type 3 constraints, suitable metrics are needed to establish termination and correctness of the derived transformation implementation: There should exist a measure $Q : \mathbb{N}$ on the state of a model, such that Q is decreased on each step of the transformation (application of a constraint to a particular domain element), and with $Q = 0$ being the termination condition of the transformation.

There are also special cases of the pattern for *entity splitting*, when the data of one source entity is used to produce the data of several target entities, and *entity merging*, when data from several source entities is used to produce the data of a single target entity.

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

Implementation: Implementation can be either by the phased creation or recursive descent implementation patterns (Section V). For phased creation the constraints can be individually implemented as phases, with different strategies being used for each type of constraint.

Individual constraints *Cn*:

$$\forall s : S_i \cdot SCond \ implies \ \exists t : T_j \cdot LPost \ and \ GPost$$

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

Constraint type	*Implementation choice*
Type 1 constraint	Approach 1: single for loop `for` $s : S_i$ `do` $s.op()$
Type 2 constraint	Approach 2: while iteration of for loop.
Type 3 constraint	Approach 3: while iteration of search-and-return for loop

Table I
DESIGN CHOICES FOR CONSTRAINTS

Code examples: A large example of this approach for a migration transformation is in [9]. The UML to relational mapping is also specified in this style in [10].

A simple example of the pattern is the specification of the three-cycles graph analysis in Section IV-C.

B. Recursive form

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

Forces: Useful whenever a platform-independent specification of a transformation is required, and the conjunctive-implicative form is not applicable, because an explicit description of the transformation relation as a single relation between the source and target models cannot be defined.

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

$$\exists s : S_i \cdot SCond_{i,j} \ and \ \exists t : T_{i,j} \cdot Post_{i,j}$$

where the S_i are source entities, the $T_{i,j}$ are target model entities, $SCond_{i,j}$ is a predicate on s (identifying which elements the constraint should apply to), and $Post_{i,j}$ defines the mapping $\tau(s)$ of s in terms of s, t and other mapping forms $\tau(s')$ for some s' derived from s.

There should exist a measure $Q : \mathbb{N}$ on the state of a model, such that Q is decreased on each step of the recursion, and with $Q = 0$ being the termination condition of the recursion (no rule is applicable in this case). Q is an abstract measure of the time complexity of the transformation, the maximum number of steps needed to complete the transformation on a particular model. For quality-improvement transformations it can also be regarded as a measure of the (lack of) quality of a model.

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

Implementation: The constraints can be used to define a recursive function that satisfies the specification, or an equivalent iterative form. The constraints can also be used to define pattern-matching rules in transformation languages such as ATL [6] or QVT [12].

Code examples: Many computer science problems can be expressed in this form, such as sorting, searching and scheduling. Update-in-place transformations, which usually employ a fixpoint iteration of transformation steps, can be specified using this pattern. For example, a transformation to remove multiple inheritance from a class diagram can be specified by constraints:

$$\begin{array}{l}(\exists c : Class;\ g : c.generalization \cdot \\ \quad c.generalization{\rightarrow}size() > 1 \ and \\ \quad\quad \exists a : Association \cdot a.end1 = c \ and \\ \quad\quad\quad a.end2 = g.general \ and \\ \quad\quad\quad a.multiplicity1 = ONE \ and \\ \quad\quad\quad a.multiplicity2 = ZEROONE \ and \\ \quad\quad g.isDeleted()) \ or \\ (\forall c : Class \cdot c.generalization{\rightarrow}size() \ \leq \ 1)\end{array}$$

In this case

$$\begin{array}{l}Q(smodel) \ = \\ \quad \Sigma_{c:Class \ non \ root}(c.generalization{\rightarrow}size() - 1)\end{array}$$

C. Auxiliary metamodel

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

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

Solution: Define the auxiliary metamodel as a set of (meta) attributes, associations, entities and generalisations extending the source and/or target metamodels. These elements may be used in the succedents of *Cons* constraints (to define how the auxiliary data is derived from source model data) or in antecedents (to define how target model data is derived from the auxiliary data).

Figure 2 shows a typical structure of this pattern. The auxiliary metamodel simplifies the mapping between source and target by factoring it into two steps.

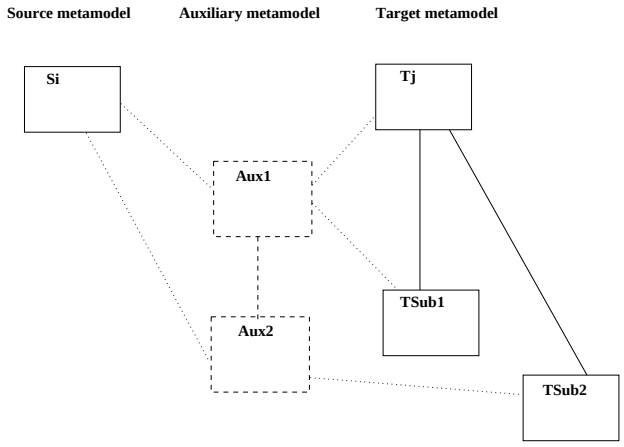

Figure 2. Auxiliary metamodel structure

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

Code example: An example is a transformation which returns the number of cycles of three distinct nodes in a graph. This problem can be elegantly solved by extending the basic graph metamodel by defining an auxiliary entity *ThreeCycle* which records the 3-cycles in the graph (Figure 3).

The auxiliary language elements are shown with dashed lines.

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

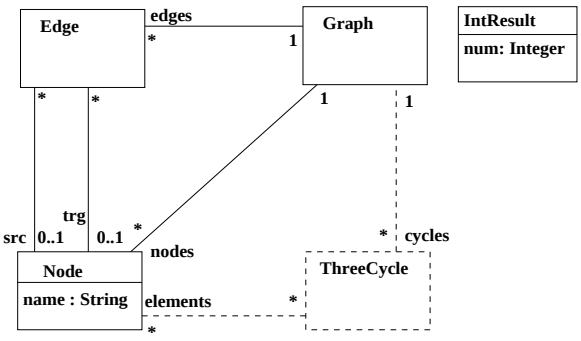

Figure 3. Extended graph metamodel

state of the transformation:

$$(C1):$$
$$\forall g : Graph \cdot \forall e1 : g.edges;\ e2 : g.edges;\ e3 : g.edges \cdot$$
$$e1.trg = e2.src \ and \ e2.trg = e3.src \ and$$
$$e3.trg = e1.src \ and$$
$$(e1.src \cup e2.src \cup e3.src) \rightarrow size() = 3 \ implies$$
$$\exists_1 \ tc : ThreeCycle \cdot$$
$$tc.elements = (e1.src \cup e2.src \cup e3.src)$$
$$and \ tc : g.cycles$$

$$(C2):$$
$$\forall g : Graph \cdot \exists r : IntResult \cdot r.num = g.cycles \rightarrow size()$$

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

Tracing is another example, which is often carried out by using auxiliary data to record the history of transformation steps within a transformation.

This pattern is referred to as *intermediate structure* in [4].

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

D. Construction and cleanup

Synopsis: To simplify a transformation specification by separating it into a phase which constructs model elements, followed by a phase which deletes elements.

Forces: Useful when a transformation needs to create and delete elements of entities. For example, because an auxiliary metamodel is being used, whose elements must be removed from the final target model.

Solution: Separate the creation phase and deletion phase into separate sets of constraints, usually the creation (construction phase) will precede the deletion (cleanup). These can be implemented as separate transformations, each with a simpler specification and coding than the single rule.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-165-6

Consequences: The pattern leads to the production of intermediate models (between construction and deletion) which may be invalid as models of either the source or target languages. It may be necessary to form an enlarged language for such models.

Code examples: An example is migration transformations where there are common entities between the source and target languages [11]. A first phase copies/adapts any necessary data from the old version (source) entities which are absent in the new version (target) language, then a second phase removes all elements of the model which are not in the target language. The intermediate model is a model of a union language of the source and target languages.

Another example are complex quality improvement transformations, such as the removal of duplicated attributes [7]. These can involve addition and removal of elements in a single step, and can be re-expressed more simply by separating these actions into successive steps.

Another implementation strategy for this pattern is to explicitly mark the unwanted elements for deletion in the first phase, and then to carry out the deletion of marked elements in the second phase.

V. Implementation Patterns

In this section we define patterns to organise the implementation of model transformations.

A. Phased creation

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

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

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

Figure 4 shows the schematic structure of this pattern.

Consequences: The stepwise construction of the target model leads to a transformation implementation as a sequence of phases: earlier phases construct elements that are used in later phases.

Implementation: The constraints are analysed to determine the dependency ordering between the target language data and entities. $T1 < T2$ means that a $T1$ instance is used in the construction of a $T2$ instance. Usually this is because

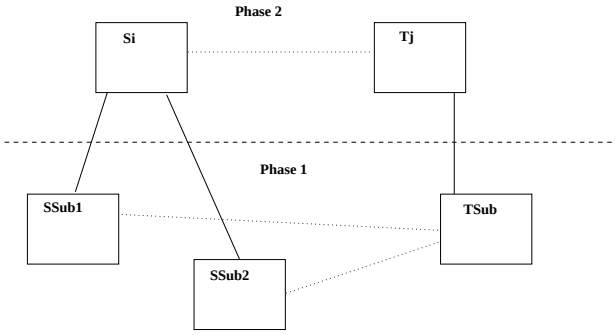

Figure 4. Phased creation structure

there is an association directed from $T2$ to $T1$, or because some feature of $T2$ is derived from an expression using $T1$ elements.

If the order $<$ is a partial order (transitive, antisymmetric and irreflexive) then the corresponding ordering of phases follows directly from $<$: a phase that creates $T2$ instances must follow all phases that create $T1$ instances, where $T1 < T2$. However, if there are self-loops $T3 < T3$, or longer cycles of dependencies, then the phases creating the entities do not set the links between them, instead there must be a phase which follows all these phases which specifically sets the links.

Code examples: The *ThreeCycle* example illustrates the simple case. Here *ThreeCycle* $<$ *IntResult*, so the phase implementing $C2$ must follow that for $C1$.

B. Unique instantiation

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

Forces: Required when duplicated copies of objects in the target model are forbidden, either explicitly by use of the $\exists_1 t : T_j \cdot Post$ quantifier, or implicitly by the fact that T_j possesses an identifier (primary key) attribute.

Solution: To implement a specification $\exists_1 t : T_j \cdot Post$ for a concrete class T_j, test if $\exists t : T_j \cdot Post$ is already true. If so, take no action, otherwise, create a new instance t of T_j and establish *Post* for this t.

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

Consequences: The pattern ensures the correct implementation of the constraint. It can be used when we wish to share one subordinate object between several referring objects: the subordinate object is created only once, and is subsequently shared by the referrers. There is, however, an additional execution cost of carrying out checks for existing elements.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-165-6

Implementation: The executable 'update form' in Java of $\exists_1 t : T_j \cdot Post$ for a concrete class T_j is:

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

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

The pattern is used in a number of model transformation languages, such as QVT-R, to avoid recreating target elements with required properties. In QVT-R it is known as the 'check before enforce' strategy.

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

C. Object indexing

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

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

Solution: Maintain an index map data structure *cmap* of type $IndType \rightarrow C$, where C is the class to be indexed, and *IndType* the type of its primary key. Access to a C object with key value v is then obtained by applying *cmap* to v: $cmap.get(v)$.

Figure 5 shows the structure of the pattern. The map *cmap* is a qualified association, and is an auxiliary metamodel element used to facilitate separation of the specification into loosely coupled rules.

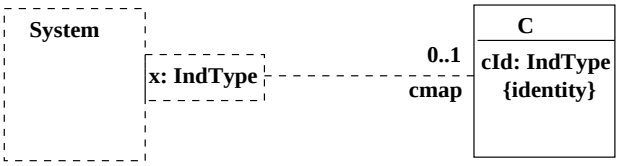

Figure 5. Object indexing structure

Consequences: The key value of an object should not be changed after its creation: any such change will require an update of *cmap*, including a check that the new key value is not already used in another object.

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

In QVT-R the pattern is implemented by defining *key* attributes by which objects can be uniquely identified.

VI. Conclusion

We have described four specification patterns and three implementation patterns, which can be used together within a development process for model transformations. These have been implemented within the UML-RSDS toolset. Other patterns which are widely used in model transformations are the *Recursive descent* pattern, where an implementation is structured as a series of recursive operations, using the hierarchical structure of source and target language entities [13].

Acknowledgment

This paper describes work carried out in the UK HoRT-MoDA project, funded by EPSRC.

References

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

[2] J. Cabot, R. Clariso, E. Guerra, J. De Lara, *Verification and Validation of Declarative Model-to-Model Transformations Through Invariants*, Journal of Systems and Software, preprint, 2009.

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

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

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

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

[7] K. Lano, *Class diagram rationalisation case study*, Dept. of Informatics, King's College London, 2011.

[8] K. Lano (ed.), *UML 2 Semantics and Applications*, Wiley, New York, 400 pages, 2009.

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

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

[11] K. Lano, S. Kolahdouz-Rahimi, *Specification of the GMF migration case study*, TTC 2011.

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

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

[14] I. Poernomo, *Proofs as model transformations*, ICMT 2008.

[15] I. Poernomo, J. Terrell, *Correct-by-construction Model Transformations from Spanning tree specifications in Coq*, ICFEM 2010.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-165-6