Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Flexible Model-to-Model Transformation Templates: An Application to ATL

Jesús Sánchez Cuadrado[a] Esther Guerra[a] Juan de Lara[a]

a. Computer Science Department,
Universidad Autónoma de Madrid,
Spain

Abstract Model transformation is one of the core techniques in Model-Driven Engineering. Many transformation languages exist nowadays, but few offer mechanisms directed to the reuse of whole transformations or transformation fragments in different contexts.

Taking inspiration from generic programming, in this paper we define model transformation *templates*. These templates are not defined over concrete meta-models, but on so-called *meta-model concepts* which later can be bound to specific meta-models. The binding mechanism is flexible as it permits mapping concepts and meta-models with certain kinds of structural heterogeneities. The approach is general and can be applied to any model transformation language. In this paper we report on its application to ATL.

Keywords Model-Driven Engineering; Model-to-Model Transformation; Reutilization; Genericity.

1 Introduction

Model-Driven Engineering (MDE) [VS06] proposes the use of models as the key assets in software development, and hence all sorts of model manipulations are needed, like model refactorings and optimizations, model simulations, model-to-model transformations and code generations. In this way, model transformations become one of the basic building blocks of MDE.

Even though MDE has been applied successfully in many scenarios, it still needs appropriate mechanisms to handle the development of complex, large-scale systems. One such mechanism is a facility to make model transformations reusable, so that they can be applied in different contexts, with different meta-models. This facility would enable the creation of transformation patterns and idioms [BJP05], as well as libraries of transformations addressing recurrent transformation problems. Some examples

Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara. *Flexible Model-to-Model Transformation Templates: An Application to ATL*. In Journal of Object Technology, vol. 11, no. 2, 2012, pages 4:1–28.
doi:10.5381/jot.2012.11.2.a4

of model manipulations commonly needed in different contexts are calculating the transitive closure of a relation, moving and merging nodes through a relation (like pulling up a method or an attribute), and cycle detection. On a larger scale, we may need to calculate the flow graph [BL93] of different variants of procedural languages with similar constructs, or to transform from several kinds of workflow languages (like YAWL, BPMN or BPEL) into Petri nets. Unfortunately, the definition of model transformations is normally a type-centric activity, in the sense that transformations are defined using types of specific meta-models, thus making their reuse for other meta-models difficult.

In this work, we bring into model transformation elements from generic programming in order to make model transformations reusable over multiple meta-models. In particular, we propose defining model transformation templates over *concepts* [GJS+06, SM09]. In generic programming, a *concept* expresses the requirements for a type parameter of a template. In our case, a concept is a meta-model that defines the set of requirements that a specific meta-model must fulfill to be used as the source or target domain of a model transformation. Once the concept is bound to a specific meta-model satisfying the concept requirements, the transformation becomes applicable to this meta-model.

In [dLG10], we proposed concepts as a mechanism to add genericity to models, meta-models and in-place transformations, and in [RGdL+11] we used them to define generic model management operations. However, these previous works only allowed a restricted kind of binding between the concepts and the meta-models consisting of an exact embedding of the former in the latter (i.e., no structural heterogeneity was allowed). In this paper, we apply concepts to model-to-model transformations, and propose a more powerful notion of binding that permits a structural mapping, a replication of elements in the concept, as well as adaptations from the structure in the concept to the structure of the meta-model. Both types of variability induce modifications in the transformation template when instantiation takes place.

As a proof of concept, we report on an implementation on top of ATL [JABK08] where the transformation templates are instantiated by a higher-order transformation (HOT). Nonetheless, our approach is general and therefore applicable to other transformation languages. Furthermore, we illustrate the usefulness of our approach with a case study consisting on the definition of a concept for procedural programming languages and a transformation template to calculate the flow graph of a program [BL93]. We show the reusability of the template by binding the concept to two meta-models developed by third-parties.

This paper extends [CGdL11] with a detailed account of the rules that the bindings in the source and target domains should fulfil to ensure behaviour preservation when using a template; a more flexible binding which interprets the inheritance relations in concepts and enables a structural mapping from concepts to meta-models; improvement of tool support; a further reuse example in the case study (a binding to PL/SQL [FP09]); and a more exhaustive comparison with related research.

The rest of the paper is organized as follows. Section 2 reviews the main elements of generic programming and outlines our approach. Section 3 introduces transformation templates, concepts and bindings. Next, Section 4 adds more flexible parameterisation capabilities to our approach by providing multiple cardinality for our concepts and adapters for the bindings. Section 5 outlines our implementation on top of ATL and Section 6 presents a case study. Section 7 compares with related work and Section 8 concludes.

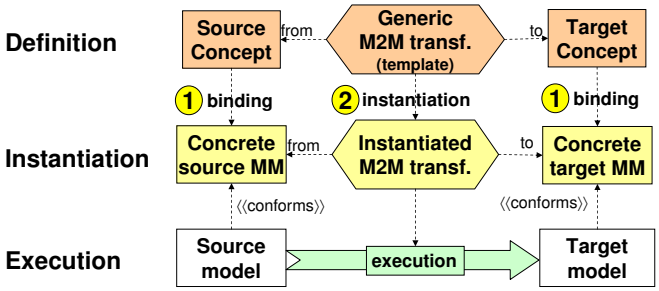


Figure 1 – General working scheme of our approach.

2 Genericity in Model Transformation

Generic programming is a programming paradigm found in many languages like C++, Haskell, Eiffel and Java [GJL+03]. Its goal is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct reuse in software construction. It involves expressing algorithms with minimal assumptions about data abstractions, as well as generalizing concrete algorithms without losing efficiency. It promotes a paradigm shift from types to algorithms' requirements, so that even unrelated types may fulfil those requirements, hence making algorithms more general and reusable [GJS+06, SM09].

Generic programming is realized through genericity mechanisms such as function or class templates in many programming languages, like C++ or Java. Templates declare a number of type parameters for a given code fragment, which later can be instantiated with concrete types. Templates can also define requirements on the type parameters, so that only those concrete types fulfilling the requirements are considered valid. A unit expressing a set of requirements is called a *concept* [GJS+06], and usually declares the signature of the operations a given type needs to support in order to be acceptable in a template. Hence, templates rely on concepts to declare the requirements needed from their type parameters.

Based on these ideas, we have defined a novel approach for generic model-to-model transformations that is outlined in Figure 1. Similar to programming templates, we build model transformation templates. These are transformations in which either the source or the target domain (or both) is not a specific meta-model, but contains *variable* types. The requirements for the variable types (required properties, associations, etc.) are specified through a concept. A concept has the form of a meta-model as well, but its elements (classes, attributes, associations) are interpreted as variables that need to be bound to elements of some concrete meta-model.

In order to use a model transformation template, a binding from its associated concepts to specific meta-models needs to be specified (step 1 in Figure 1). Once such a binding is established, our approach automatically instantiates a concrete transformation from the template (step 2), which can be executed on regular instance models of the bound meta-models. This approach yields reusable transformations because we can bind a concept to several meta-models, so that the generic transformation is applicable to all of them. Finally, although the figure assumes a generic transformation with both domains being concepts, either the source domain or the target domain can be concrete meta-models.

A crucial issue to increase the reuse opportunities of a transformation template is to

have a flexible binding mechanism allowing concepts to be mapped to a large number of meta-models. We propose two such mechanisms: cardinality annotations to enable variability on the number of bindings for a concept element, and binding adaptation to overcome structural heterogeneities between concepts and meta-models. As we will see, the more sophisticated the binding is, the more complex the instantiation of the transformation becomes. In this paper, we implement the instantiation mechanism for template transformations by a HOT over ATL transformations.

3 Concepts, bindings and templates

A *meta-model concept* is a specification of the minimal requirements that a meta-model should fulfil to qualify for the source or target domain of a generic model-to-model transformation template (or in general, of a generic model management operation [dLG10, RGdL+11]). From a practical point of view, a concept is just a meta-model and can be used as the source or target domain of a transformation template.

As an example, the upper part of Figure 2 shows the definition of a transformation template that creates a Java model from any object-oriented (OO) system. In this case, the domain *from* is specified by a concept, whereas the domain *to* is a concrete meta-model (a simplification of the Java meta-model). The transformation template, shown to the right with ATL syntax, is made of two rules. The first one *class2jclass* generates a Java class from each OO class, setting the Java class parent as the first available superclass (hence eliminating multiple inheritance). The second rule creates a Java field from each attribute. Thus, our transformation templates are just ordinary transformations where the queried or created types may be interpreted as variables instead of as types of a concrete meta-model.

In order to execute a transformation template, we have to map or bind the concepts used by the template to specific meta-models. In the simplest case, if the concept contains no inheritance relation, the binding establishes a 1-to-1 correspondence from each class in the concept to a class in the bound meta-model, from each attribute in the concept to an attribute in the meta-model, and from each association in the concept to an association in the meta-model.

In addition, concepts may include inheritance relations between classes with the purpose of simplifying the design of the concept (e.g., declaring common attributes in a parent class) or the associated transformation template (e.g., defining a single transformation rule for a parent class instead of a rule for each child). These inheritance relations are not strictly necessary in the bound meta-models, provided that the flattened structure extracted from the concept is preserved in the meta-models. Therefore, there is no need to bind the inheritance relations appearing in the concepts, as our binding is actually a structural mapping, in the line of structural subtyping in programming languages [CW85]. Hence, conceptually, the binding is a mapping from a *flattened* version of the concept (with copies of attributes and references from parent to children classes) to a specific meta-model.

We adopt the convention that only the non-abstract classes in the concept need to be mandatorily mapped, as well as their attributes and references, both owned and copied from the parents. Providing a mapping for abstract classes is optional. Thus, a concept should not contain abstract leaf classes, since an abstract class in a concept is only considered a fine-grained reuse mechanism. Moreover, a copied (i.e., inherited) attribute or reference does not need to be mapped in a child class if it has already been

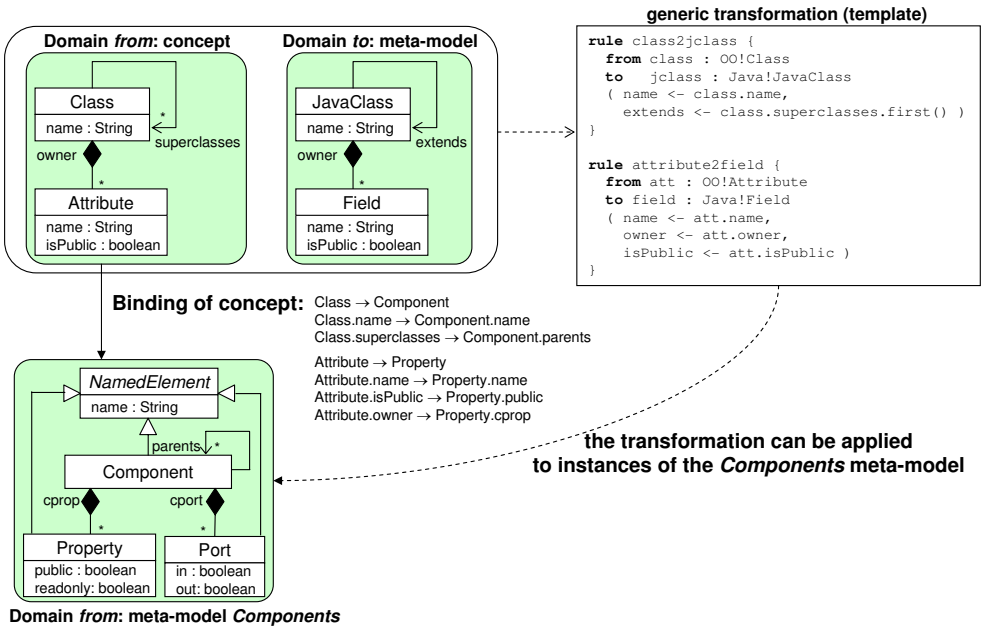


Figure 2 – Transformation template from a concept for OO systems into a fixed Java meta-model, and binding of the concept to a particular meta-model for components.

mapped in the parent. Once the binding is established, our instantiation mechanism modifies the transformation template, e.g., replicating rules and expressions, in order to make it applicable to the bound meta-model.

The binding also imposes some additional conditions to ensure the compatibility of the mapped associations and attributes, and to guarantee that the operations performed in the concept can be performed in the bound meta-model types. These conditions are different depending on whether the concept represents a domain that is being read during the transformation or whether it is being modified. We will use the term *read-enabled domain* to refer to a domain that can be read or queried (e.g., a model can be read within a transformation irrespectively of being a source or a target model), and *write-enabled domain* to refer to a domain that can be written or populated. In the following, we present binding constraints that apply to: (i) any domain, (ii) read-enabled domains and (iii) write-enabled domains. For example, if a domain is read-only, then the binding of the associated concept must fulfil the constraints (i)+(ii). If a domain is both read- and write-enabled, then the constraints (i)+(ii)+(iii) apply. In the implementation for ATL that we provide in Section 5, the source domain is read-only and the target domain is write-only.

Table 1 lists the conditions that apply to bindings of concepts in any type of domain, no matter if it is read-enabled or write-enabled. In particular:

1. **Features.** The container classes of two mapped features (attributes or references) must be bound. There is an exception though: it is allowed to map a feature to an inherited feature (i.e., a feature defined in a superclass). For instance, in Figure 2, attribute name of Class is bound to attribute name defined in NamedElement, although Class is not mapped to NamedElement. Nonetheless this is allowed because Class is bound to Component, of which NamedElement is a superclass.

	Scope	Condition
1	features	Given a class A ∈ concept, and a feature f ∈ A.features, then bind(f) ∈ X.features for some class X ∈ bind(A).ancestors
2	subtyping	Given two distinct classes A,B ∈ concept s.t. bind(A), bind(B) are defined, then A ∈ B.ancestors ⇒ bind(A) ∈ bind(B).ancestors ∧ A ∉ B.ancestors ⇒ bind(A) ∉ bind(B).ancestors

Table 1 – Conditions for bindings of both read-enabled and write-enabled concepts. Function X.ancestors returns the set of ancestors of class X, including X itself. X.features returns the set of attributes and references defined in class X.

	Scope	Condition
3	composition	Given a composition reference r ∈ concept, then bind(r) is a composition
5	type of attributes	Given an attribute a ∈ concept, then bind(a).type ⊴ a.type
6	type of references	Given a reference r to a class A ∈ concept, then bind(r).target ⊴ bind(A)
7	cardinality	Given a reference r ∈ concept, then bind(r).mincard ≥ r.mincard ∧ r.maxcard<>* ⇒ bind(r).maxcard ≤ r.maxcard

Table 2 – Conditions for bindings of concepts in a read-enabled domain. Function ⊴ is the subtype relation.

2. **Classes and subtyping.** Only the non-abstract classes in a concept need to be bound into classes of a meta-model. For the abstract classes, providing a binding is optional. If given, the binding must preserve any subtyping relation defined in the concept, so that, if two classes in the concept are directly or indirectly related by an inheritance relation, so must be the mapped classes in the bound meta-model. Conversely, if the concept does not relate two classes through inheritance, then the mapped classes in the meta-model cannot be related by inheritance either. Moreover, we allow two or more classes related by inheritance in the concept to be mapped into a single one in the meta-model (i.e. collapsing an inheritance chain). This is possible by mapping two or more classes in the concept to the same class, and mapping all owned and inherited features of the classes in the concept to owned or inherited features of the class in the concrete meta-model in order to preserve the structure. These conditions will be relaxed in Section 4 by allowing non-abstract classes not to be mapped or to be mapped more than once.

Next, we describe further conditions that are different depending on whether the concept is read-enabled or write-enabled. These conditions are also summarized, using a formal syntax, in Tables 2 and 3. The first column of the tables refers to the number assigned to the conditions in the following enumeration:

3. **Composition.** In a read-enabled domain, a composition in a concept cannot be mapped into a non-composition reference in the meta-model. This is so as models

	Scope	Condition
3	composition	Given a non-composition reference $\mathsf{r} \in \mathsf{concept}$, then $\mathsf{bind(r)}$ is a non-composition
4	abstract classes	Given a class $\mathsf{A} \in \mathsf{concept}$ s.t. $\mathsf{A.abstract{=}false}$, then $\mathsf{bind(A).abstract{=}false}$
5	type of attributes	Given an attribute $\mathsf{a} \in \mathsf{concept}$, then $\mathsf{a.type} \trianglelefteq \mathsf{bind(a).type}$
6	type of references	Given a reference r to a class $\mathsf{A} \in \mathsf{concept}$, then $\mathsf{bind(A)} \trianglelefteq \mathsf{bind(r).target}$
7	cardinality	Given a reference $\mathsf{r} \in \mathsf{concept}$, then $\mathsf{bind(r).mincard} \leq \mathsf{r.mincard} \wedge$ $\mathsf{bind(r).maxcard{<>}*} \Rightarrow$ $(\mathsf{r.maxcard{<>}*} \wedge \mathsf{bind(r).maxcard} \geq \mathsf{r.maxcard})$
8	unbound references	Given a class $\mathsf{A} \in \mathsf{concept}$ s.t. $\mathsf{bind(A)}$ is defined, then $\forall$ reference $\mathsf{r}_1 \in \mathsf{bind(A).references}$ s.t. $\mathsf{r}_1\mathsf{.mincard}{>}0$ $\exists$ reference $\mathsf{r}_2 \in \mathsf{concept}$ s.t. $\mathsf{bind}(\mathsf{r}_2){=}\mathsf{r}_1$

Table 3 – Conditions for bindings of concepts in a write-enabled domain. Function $\trianglelefteq$ is the subtype relation.

cannot contain cycles of compositions, and therefore a transformation template may not expect them when performing a query over the models. However, an instance of the bound meta-model may contain cycles through non-containment references, which may cause the transformation to behave unexpectedly, for instance if a query navigates these references recursively and the queried model has cycles.

In a write-enabled domain, a non-composition reference in the concept cannot be mapped into a composition in the meta-model. This is so as the transformation template may create cycles, which is not problematic in the concept as the reference is a non-composition, but may be problematic in the meta-model as the mapped reference is a composition.

In practice, these conditions are too restrictive for transformation templates that do not create cycles and for bindings that would not provoke any misbehaviour. Thus, it is possible to relax these conditions by generating pre-conditions and post-conditions that guarantee, for the features involved, that the source model does not contain cycles, and that no element is added more than once to containment references when populating the target model. Thus, the user of the transformation template is allowed to overlook these conditions if the input model satisfies the generated pre-conditions and the target model satisfies the post-conditions. This strategy ensures that there are no cycles in the source or the target that violate the binding conditions with respect to compositions.

4. **Abstract classes.** Providing a mapping for the abstract classes in the concept is optional. In read-enabled domains, there is no constraint that forbids mapping abstract and concrete classes, i.e., we can map an abstract class in the concept to either an abstract or a concrete class in the meta-model, and we can map a concrete class in the concept to either an abstract or a concrete class in the meta-model.

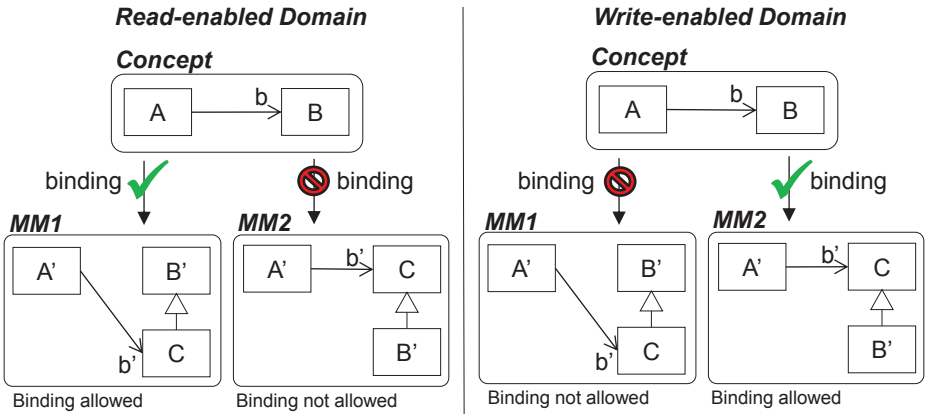


Figure 3 – Rules for binding references, according to the type of the reference ends.

In write-enabled domains, however, it is not allowed to map a concrete class A in the concept to an abstract class A' in the meta-model. The reason is that, if the transformation template creates A objects, there will be an error as A' is abstract and therefore cannot be instantiated.

5. **Type of attributes.** In a read-enabled domain, attributes with a primitive type can be mapped to attributes of the same type or a subtype. The reason is that, conceptually, a binding of an attribute x in the concept to an attribute x' in the source meta-model induces an assignment $x := x'$. The only way to make this assignment work without resorting to casting operations is to require type substitutability: the type of x' has to be the same type or a subtype of x (or to be possible to coerce it). For example, we can map a `double` attribute in the concept to an attribute of types `double`, `float`, `long` and `int` in the source meta-model, but we cannot map an `int` attribute to a `double` attribute.

   In a write-enabled domain we have the converse situation, so that attributes can be mapped to attributes with the same type or a supertype. Therefore, while for write-enabled domains we have covariant conversion of attributes, for read-enabled domains we have contravariance. As we will see, a similar situation happens with the types of references.

6. **Type of references.** In a write-enabled domain, if a concept contains a reference to a class B, then the bound reference in the meta-model must point to the class bound to B or to a supertype of it (see Table 3). The right of Figure 3 illustrates this issue, where the primes indicate the binding (e.g., A is mapped to A' and so on). The rationale is that if a transformation template connects A and B objects, performing the binding to the meta-model MM1 will produce an error because the b' reference cannot store B' objects as expected. The binding to MM2 does not raise errors because, in this case, b' can store C and B' objects.

   In a read-enabled domain we have the opposite situation, as the left part of Figure 3 shows (see also Table 2). In this case, the binding for the meta-model MM1 is allowed because navigating through b' leads to C, which is subtype of B' and therefore inherits all its features. The binding for MM2 is not allowed in general because navigating through b' leads to C, which may not define all features queried from B'.

7. **Cardinality.** In a write-enabled domain, a transformation template creates instances of the meta-model bound to the concept. Therefore, the binding needs to ensure that any possible instance of the concept (interpreted as a meta-model) is also a valid instance of the bound meta-model. For this reason, it is not possible to map a reference in the concept to a reference with higher lower bound or lower upper bound, i.e., only the same or wider cardinality intervals can be mapped.

   Conversely, any query over elements of a concept in a read-enabled domain should be possible over the elements in the bound meta-model. Hence, a reference in the concept can only be mapped to a reference with the same or narrower cardinality interval in the meta-model, as we require that any instance of the bound meta-model can be seen as a valid instance of the concept. As a consequence, domains which are both read- and write-enabled require each reference in the concept to be mapped to a reference in the meta-model with the same cardinality interval.

   These conditions may need to be adapted depending on the features of the specific query language used. For instance, as we will show in Section 5, in OCL the syntax for accessing multivalued and monovalued references is different, which makes the conditions more restrictive.

8. **Unbound references.** In a write-enabled domain, if a class A in the concept is bound to a class A' in a meta-model, then A' cannot own (or inherit) a mandatory reference to which no reference in the concept is bound. The reason is that the transformation might be creating A' instances with missing mandatory references, leading to the creation of incorrect models. Such a restriction is not necessary in read-only domains.

The binding is conceptually a function from the set of classes, associations and attributes of the *flattened* concept. Hence, we cannot map one element in the concept to two elements in the meta-model. Nonetheless, it is possible to map several elements in the concept to the same element in the meta-model (i.e., having non-injective bindings) whenever this follows the constraints enumerated before. In addition, not all elements in the meta-model need to receive a binding from the concept (i.e., non-surjective binding).

As an example, Figure 2 shows a valid binding from a concept modelling the requirements of the source domain of a transformation (an OO system), to a particular meta-model for defining components. The source domain is assumed to be read-only. The binding maps classes Class and Attribute in the concept to classes Component and Property in the meta-model, associations superclasses and owner (a composition) to parents and cprop (a composition as well), attribute name of Class to the inherited attribute name in the component, and attributes name and isPublic in class Attribute to attributes name and public in class Property. In the meta-model, class Port, its owned features and the cport reference do not receive any binding. Once the binding is established, the transformation can be applied to instances of the bound meta-model, creating a Java class for each component in the meta-model instance.

From the point of view of the model transformation engine builder, there are two ways to apply a transformation template to the bound meta-models. The first possibility is to encode a HOT that takes as input the template, the bound meta-models and the binding, and produces a transformation that is directly applicable to the meta-models. In the template, the HOT replaces each class, association and attribute declared in the concept by elements in the concrete meta-models, as specified by the

binding. It may also need to perform further rewriting (e.g., replicating rules) in order to account for structural heterogeneities between the concept and the meta-model (e.g., when an abstract class in the concept remains unbound). For instance, the transformation generated from our example would use components and properties, but no OO classes anymore. In this way, the resulting transformation can be executed by a regular transformation engine. This is the approach we have taken in this paper. The second possibility is leaving the transformation unmodified, and including a level of indirection managed by the transformation engine. In this case, when a generic transformation is executed for a given binding and the engine finds an element defined by a concept, then it has to go through the binding to obtain the bound type. This approach is followed by tools such as [dLG10]. An advantage of the HOT-based approach followed in this paper is that there are not performance penalties due to adaptations made at runtime. However, the implementation of this approach is more complex, since some static analyses are needed. Besides, it requires an extra step to generate the adapted transformation.

4 Adding flexibility to concepts and bindings

The binding mechanism presented so far provides a structural mapping between the elements in the concept and those in a particular meta-model, which may overcome certain heterogeneities concerning inheritance hierarchies between them. However, in practice, one may require more flexibility in order to increase the potential for reuse. For example, in Figure 2 we may also want to treat Ports as Attributes in order to generate Java fields for them. However, we defined the binding as a function and, therefore, an element in the concept cannot be bound to several meta-model elements. In Section 4.1 we will show how to extend a concept with an interval for the cardinality of its elements so that they can be replicated and subsequently bound to more than one element in the meta-models.

Once we can bind Attribute to Port (in addition to Property), we should map attribute isPublic to some attribute in Port. However, Port does not define any meaningful attribute modelling visibility. In Section 4.2 we will show how to use binding adapters to overcome this problem. An adapter is an expression evaluated in the context of the bound meta-model, which returns a suitable element as the target of a binding. In this way, an adapter could provide a binding of isPublic to the value *true*. In general, adapters resolve structural heterogeneities between concepts and meta-models, in the style of [WRK+10].

4.1 Cardinality of concept elements

Our previous binding definition requires each element in the concept to be mapped to at most one element in the bound meta-model: abstract classes in the concept may remain unbound, but the rest of elements must be mapped to exactly one element in the meta-model (1-to-1 binding). However, in practice, we sometimes find that a class in a concept is modelled by several classes in the meta-model. If all these classes have a common parent, then the binding can be performed as usual by mapping the parent. The problem arises when the classes in the meta-model do not share a common parent, or when there is a common parent but it has more children than those we would like to bind, so that mapping the parent is not a suitable solution.

In order to solve this problem, we assign a cardinality to leaf classes and associations

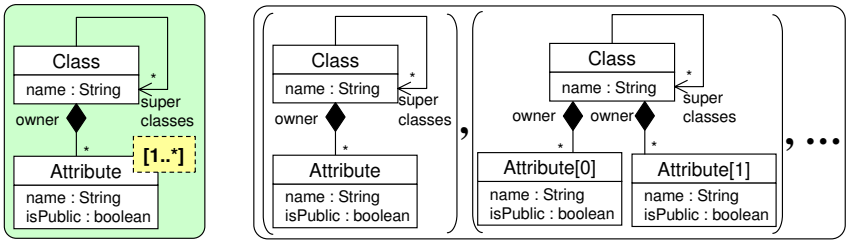


Figure 4 – Annotating a concept with cardinality (left). Set of its expansions (right).

in concepts. In this way, those elements in the concept that are allowed to be bound to one or more elements in the meta-model must define a cardinality interval "1..*" (1-to-n binding). It is also possible to specify a cardinality interval "0..1" or "0..*" if the concept element can remain unbound (i.e., it can be mapped to no meta-model element). In general, the cardinality of an element is transformation-specific, that is, the transformation template developer must annotate a concept explicitly to enable its usage. In ATL-like languages and in our current implementation, the cardinality is restricted to be exactly 1 for classes of target concepts and for associations of source concepts. Please note that we only allow cardinality annotations on classes with no children, due to the conflicts that could arise when parent classes are replicated, but not their children.

Intuitively, a concept containing elements with cardinality annotations is equivalent to a (possibly infinite) set of "flat" concepts where the cardinality of all elements is 1. Similar to [BGdL10], the set of flat concepts is calculated by performing all possible expansions in the cardinality interval of every element in the concept. If the cardinality interval includes 0, then one of the possible expansions implies deleting such element. Figure 4 illustrates this technique. To the left, class Attribute in the concept has been annotated with its cardinality, whereas elements without annotations have a cardinality 1. Thus, we can replicate class Attribute together with the associations in which it participates an unbounded number of times. To the right, the figure shows the unfolded concepts without cardinality intervals in case we replicate Attribute once or twice. The concrete number of replicas must be indicated when binding the concept to a particular meta-model. For instance, in order to map Attribute to both Property and Port in the components meta-model shown in Figure 2, we must select two replicas. Then, one can perform a 1-to-1 binding from the corresponding expanded concept to the meta-model.

Choosing a cardinality different from 1 for a concept element induces an adaptation of the associated transformation template. If a class in the concept is mapped to more than one element, then the rules defined for this class should be replicated for each mapping. In our example, rule *attribute2field* would be replicated twice: one with class Port in the from domain, and another one with class Property. If an element is not bound, then the rules defined for this class are deleted. If the cardinality is defined for an association, then the instructions assigning a value to the association should be replicated for each specified binding (or deleted if it is unbound). For instance, if the target domain in Figure 2 were a concept and we map relation owner to two associations, then the second line in the body of rule *attribute2field* would be duplicated.

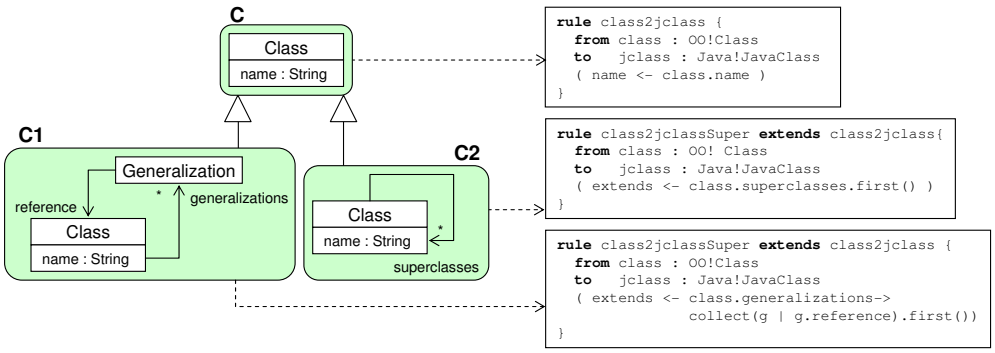


Figure 5 – Binding concepts to concepts, seen as a subtyping relation between concepts.

4.2 Binding adapters

A concept expresses requirements for a transformation. However, it also reflects a design decision which could be realized differently. For instance, the inheritance relation superclasses in the concept of Figure 2 could be implemented with an intermediate class, as in the case of the UML meta-model [OMG], whereas the isPublic attribute could also be an enumerate, or a boolean with the opposite meaning (e.g., isPrivate). Since we do not want to define different concepts and templates for each design possibility, and in order to provide more reuse opportunities for a given template, a mechanism to overcome such heterogeneities is desirable.

A first solution is to resort to subtyping relations between concepts [SJ07]. In this case, the commonalities of the different solution concepts are extracted to a supertype concept, which can be extended in order to provide alternative solutions for specific fragments. This can be seen as a 1-to-1 binding from the general concept to its extensions. Figure 5 shows our example expressed in this way. The parent concept C only includes the class and its name, and children concepts $C1$ and $C2$ provide two alternatives to express the inheritance. This solution implies a fragmentation of the transformation template as well, and relies on a composition/extension mechanism in the transformation language. In the figure, both concepts $C1$ and $C2$ extend the template defined in C. Hence, this way of reuse has the drawback that the developer has to foresee all possible extensions, and the template has to be built in such a way that it can be extended, which sometimes can be difficult or undesired. Moreover, not all transformation languages support rule extension.

For this reason, we have devised a more flexible mechanism that does not impose restrictions on the way concepts and templates are built. In this case, one of the possible concepts is chosen, and only one template is defined accordingly. When the template is instantiated for a particular meta-model, if the structure of the concept and the meta-model differ, it is possible to build an *adapter* to fix the heterogeneity. An adapter is an expression evaluated in the context of a bound meta-model type, returning a value (a primitive type or a reference type) that is assigned as binding for some attribute or association in the concept. It is important to note that adapters are only applicable to source concepts.

Figure 6 illustrates this solution. It shows the binding of our concept to a UML meta-model where inheritance is represented by an intermediate class. To solve this heterogeneity, the binding of the superclasses association is given by an OCL expression that returns a suitable value. In this way, the adapter induces a modification in the

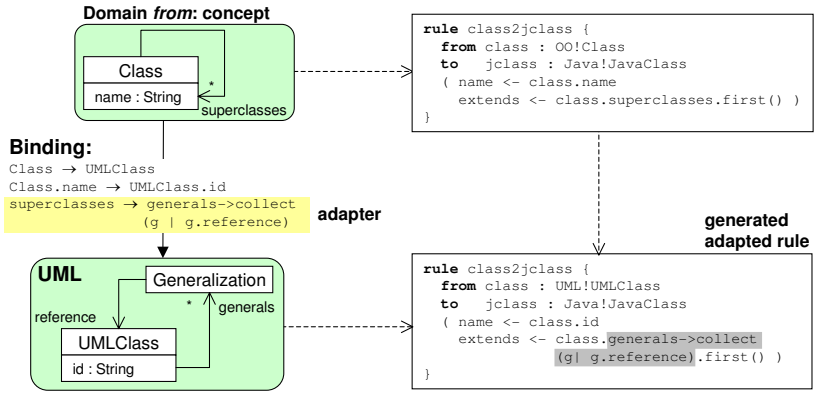


Figure 6 – A binding adapter (left). Semantics of adapter as template modification (right).

transformation template so that each reference to superclasses is replaced by the adapter expression. Note how this solution is non-intrusive as it does not require modifying the bound concrete meta-models, which in some cases may not be possible.

An additional benefit of adopting adapters to solve heterogeneities is that, as many adaptations are recurrent, we can build libraries of reusable common adapters. These can be implemented using genericity as well, defining them atop generic types that are bound to the concept and meta-model of the binding that needs to be adapted. We are currently creating a categorization of adapters to resolve commonly occurring heterogeneities. For instance, one generic adapter we have identified in this categorization is called *association to intermediate class association*, which permits mapping directly superclasses to Generalization in Figure 6.

Hence, a common strategy to develop reusable transformations is to use concepts as simple as possible. In this way, the transformation templates become simpler and easier to understand. Then, adapters can be used to reuse such transformation templates for particular meta-models. We believe this strategy will enable the development of reusable transformation patterns and idioms, which later can be customized for particular usages.

5 Tool Support

We have implemented a prototype to support our approach to model transformations templates. It currently targets ATL. In this section we first explain a DSL for expressing bindings, and then we describe the instantiation mechanism we have built.

5.1 Bindings

In our tool, a concept is defined as an Ecore meta-model [SBPM08], and its elements can be annotated with the allowed cardinality. A binding between a concept and a meta-model is represented as a model. We have created a textual concrete syntax (using TCS [JBK06]) to describe such bindings, and use OCL to define adapters.

As an example, Listing 1 shows the binding presented in Figure 2 expressed with our concrete syntax. Class is mapped to Component (line 2), and Attribute is mapped to both Property and Port (line 3). This mapping is allowed due to the cardinality

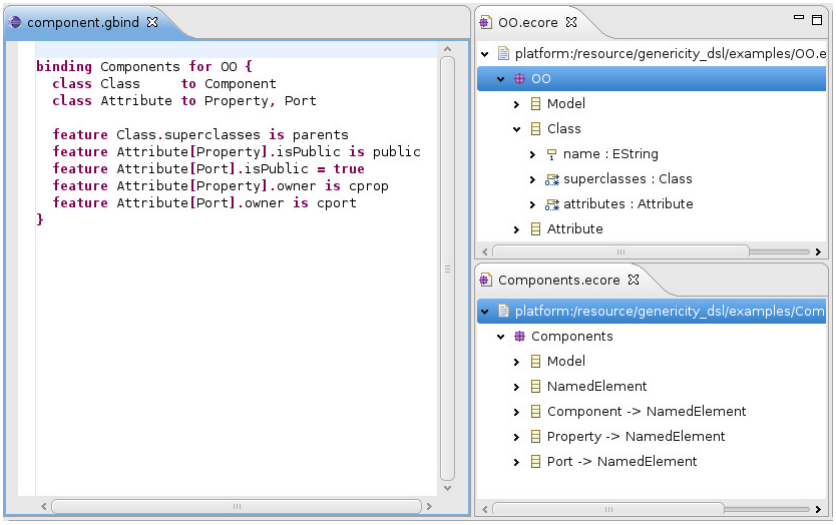


Figure 7 – Snapshot of the tool.

interval associated to Attribute which implicitly expands twice the class. The superclasses property is naturally mapped to parents (line 5). However, mapping the Attribute.isPublic and Attribute.owner properties requires specifying a context, as Attribute is bound to two classes. In the listing, the property is mapped to public in the case of Property (line 6), and to an OCL expression (i.e., an adapter) in the case of Port (line 7). Additionally, the NONE keyword allows a concept element to be unbound. This may imply removing some transformation rules in the adapted transformation, so it is responsibility of the generic transformation developer to annotate the metaclass with a minimum cardinality of 0. Finally, our tool also allows some mappings to be implicit when a class or feature has the same name in the concept and in the meta-model. For instance, we do not need to map Class.name to Component.name. Also, abstract classes do not need to be mapped.

```
1  binding Components for OO {
2    class Class to Component
3    class Attribute to Property, Port
4
5    feature Class.superclasses is parents
6    feature Attribute[Property].isPublic is public
7    feature Attribute[Port].isPublic = true
8    feature Attribute[Property].owner is cprop
9    feature Attribute[Port].owner is cport
10 }
```

Listing 1 – Binding a concept to a meta-model.

Additionally, it is possible to define attribute helpers for the meta-model, which override operations attached to concept classes. In this way, transformation templates can include queries with "holes" that will be filled differently by each meta-model.

Figure 7 shows a snapshot of our tool being used to specify the binding for this example. As can be seen, a concept is defined as a regular Ecore meta-model (OO.ecore), whereas the binding model between the concept and the meta-model (Components.ecore) can be written in a dedicated editor using a concrete textual syntax (component.gbind).

5.2 Templates in ATL

We support transformation templates written in ATL, which has been chosen as the primary language for our experiments because of three reasons. First, it is a hybrid model-to-model transformation language that clearly separates the declarative part and the imperative part, which allows us to focus on the declarative part. Secondly, the source domain is read-only and the target domain is write-only, which allows bindings with greater flexibility. Finally, ATL is defined through a meta-model and transformations are represented as models, facilitating the definition of the HOT that we need in order to instantiate transformation templates.

Given a template and a binding from the participating concepts to some meta-models, a HOT is in charge of instantiating the template, replacing generic types by those in the meta-models and performing further rewritings as we explain next. So far, we support the declarative part of ATL. The modifications to the original template depend on whether the concept is for the source or target domains, and on the chosen cardinality for each particular concept element. The following rules are applied in the case of binding elements of a source concept:

- *Class with cardinality 1.* Each usage of the concept class is renamed to its bound class.
- *Class with a binding cardinality >1.* We have identified several cases where it is possible to safely instantiate the original template. Each ATL construct requires a different strategy:
  - *Matched rule.* A new copy of the rule is created for each bound class, where the name of the latter replaces the one of the original concept class.
  - *Helper.* A new copy is created for each bound class. The context is replaced accordingly to the bound class. If a parameter type is bound to several classes, it is replaced by OclAny. The current version of ATL only checks the existence of features at runtime, so there is no need to perform further adaptations.
  - *Lazy rule.* A new copy is created for each bound class. Explicit calls to the original lazy rule are replaced with an OCL expression that checks the type of the source element in order to call one of the new lazy rules. For instance, the expression thisModule.myLazyRule(obj) is replaced by the expression in Listing 2, if the type of obj has been mapped twice.

```
1  if obj.oclIsKindOf(ConcreteMetaclass1) then
2      thisModule.myLazyRule_for_ConcreteMetaclass1(obj)
3  else
4      if obj.oclIsKindOf(ConcreteMetaclass2) then
5          thisModule.myLazyRule_for_ConcreteMetaclass2(obj)
6      endif
7  endif
```

Listing 2 – OCL expression that replaces an invocation to a lazy rule.

  - *allInstances.* Each occurrence of ConceptMetaclass.allInstances is replaced by ConcreteMetaclass1.allInstances.union(ConcreteMetaClass2.allInstances).union(...).
  - *oclIsKindOf.* Each occurrence of obj.oclIsKindOf(ConceptMetaclass) is replaced by obj.oclIsKindOf(ConcreteMetaclass1) or obj.oclIsKindOf(ConcreteMetaclass2). The same applies to oclIsTypeOf.

- *Class with cardinality 0.* If a class is mapped to none, the following rewritings are applied:
  - *Matched rule.* The rule is safely deleted, because ATL does not fail with unresolved bindings.
  - *Helper.* Every helper for that class is deleted.
  - *Lazy rule.* It is deleted. Every call to the rule is replaced by OclUndefined.
  - *allInstances.* Each occurrence of ConceptMetaclass.allInstances is replaced by an empty collection.
  - *oclIsKindOf.* Each occurrence of obj.oclIsKindOf(ConceptMetaclass) is replaced by false.
- *Feature and binding adapters.* Each usage of a concept feature is renamed to the bound feature, or in case a binding adapter is used, replaced by the adapter's OCL expression. Since this requires typing information that is not provided by the ATL compiler, we rely on ATL/OCL attribute helpers. Thus, for each concept feature, a new attribute helper with its name is attached to the bound class. If several bindings are specified for a class, then several helpers are created. The helper's body is either the name of the concrete feature or the OCL expression of the adapter. In this way, the rules in the instantiated template use the names of concept features, and the ATL engine selects the appropriate attribute helper through virtual dispatch.
- *Non-mapped abstract class.* If an abstract class in the concept is not mapped to a class in the meta-model, and the abstract class is used in the transformation template (e.g., in the *from* part of a rule), then the adaptation rules are similar to that of *class with cardinality* > 1, but mapping the abstract class to the corresponding mappings of all its leaf subclasses.
- *Helpers.* A helper defined in the binding is translated to a new ATL helper, appended to the transformation.

For target concepts, we apply the following rewriting rules to the templates:

- *Class.* Each usage of the concept class is renamed to the bound class. The cardinality of target classes must be always 1.
- *Feature with cardinality 1.* The feature in the left part of an ATL binding is renamed to the bound feature.
- *Feature with cardinality >1.* The ATL binding is replicated as many times as the cardinality indicates, and then the features are renamed as specified in the binding.
- *Feature with cardinality 0.* The ATL binding is removed.

Our binding adapters are only possible for source concepts, because the creation of the target model in ATL is driven by the source model (i.e., rule matches and queries), and therefore there is little flexibility to adapt the structure of the target model.

Dealing with a mismatch between the cardinality of a feature in the source concept and its counterpart in the meta-model poses, in OCL, the problem of the different

syntax and operations for handling monovalued and multivalued features. For instance, while it is conceptually valid to map a 1..* reference in a source concept to a 1..1 reference in the meta-model, the adapted transformation would fail because it expects to deal with an OCL collection. The solution is to write an adapter that wraps the single element pointed by the reference into an OCL collection, so that it becomes compatible. This adaptation has to be done manually now, but can be automatically supported by our HOT. In the case of target concepts there is no problem because an ATL binding transparently converts a single element (the right part) into a collection (a multivalued feature in the left part).

Our current implementation has a couple of limitations, though. The first one is related to binding adapters that adapt a target feature whose name is the same as the concept feature. The problem arises because we create an attribute helper with the same name as the concept feature. This overrides the original feature, which indeed causes a recursive call when the same feature name is used in the helper's body. Unfortunately, as far as we know, there is no way in ATL to obtain the feature value without calling the attribute helper. A workaround is to add an underscore to feature names in the concept, so that name clashes are unlikely to occur.

The second limitation is that we do not support non-injective bindings (map several elements in the concept to the same element in the meta-model). The problem is that if two types in the concept are used as source type of two different rules, when mapped to a single element will produce two rules with the same source type, which may provoke a conflict in ATL. Nonetheless this limitation is specific to ATL, and would not cause problems in other languages, especially in those based on graph transformation.

Finally, it is worth noting that our approach is not limited to ATL, but it could be adapted to other transformation languages. In particular, declarative transformation languages such as QVT Relational [OMG05] and Tefkat [LS06] explicitly establish how target elements relate to source elements, which facilitates rule adaptation. However, the integration of adapter code (written in OCL) will be easier in QVT than in Tefkat because the former uses OCL (i.e., the translation is straightforward) while the latter is based on a logic-based pattern matching language. Another requirement for our approach is to have the abstract syntax of the transformation language available as a model, so that it can be transformed with a HOT.

6 Case Study

This section illustrates the applicability of our approach in a non-trivial scenario, namely a generic transformation to compute the *flow graph* for a family of languages with imperative constructs. Flow graphs are widely used in program analysis to describe the control flow of imperative programs. Given the flow graph of a procedure, several analysis and further transformations are possible, such as visualizing the structure of the algorithm implemented by the procedure, detecting unreachable code or invalid "configurations" (e.g., jumping within a loop), computing the cyclomatic complexity of a procedure, or performing some program profile analyses [BL93].

A naive approach to tackle this scenario would be to build specific transformations from each particular procedural language into flow graphs. Instead, since all these languages share common features that can be included in a concept, we prefer to build just a unique transformation template defined over such a concept. In this way, the template will be applicable to every procedural language for which we can bind its meta-model to the concept.

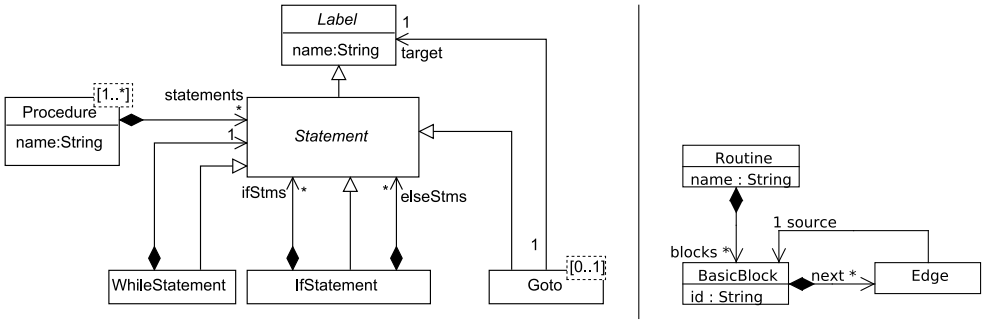


Figure 8 – Concept for imperative languages (left). Concept for control flows (right).

Figure 8 shows to the left the concept used by our generic transformation as source domain. It includes standard control statements commonly found in many imperative languages. The class Procedure is used to model both functions and procedures, and hence it has been labelled with "1..*" cardinality to accommodate the specific modularity notions of different languages. A procedure is made of a sequence of statements. Statements may be annotated with a label, and are refined into typical control instructions: *if*, *while* and *goto*. No further statements are needed, as the flow graph transformation only deals with control statements. In addition, the Statement class contains the toInfo operation that has to be implemented by the binding to a concrete meta-model. Its purpose is allowing the transformation to annotate the flow graph with precise information about the statements, so that, e.g., it becomes useful for visualization.

Our transformation template implements a variation of the algorithm proposed in [ASU86], based on partitioning a piece of code into basic blocks. A *basic block* is "*a sequence of consecutive statements where the execution flow can only enter the basic block through the first instruction in the block and leave the block without halting or branching*". We omit the details of the implementation for the sake of simplicity but it is worth noting that this is not a straightforward ATL transformation, comprising 7 rules and 13 helpers. Basically, there is one rule transforming every leader statement into a basic block, and another rule for each kind of control statement (e.g., *if*). Model navigation code involves most of the complexity of the ATL transformation template, for instance to determine which statement is a leader, its followers, etc. Listing 3 shows a simplified excerpt of the transformation. The isLeader helper checks whether a statement is leader of a basic block. The statement2basicblock rule deals with every non-control statement that acts as leader of a basic block and establishes the edges to the next basic blocks (e.g., call to the goto_statement2edge lazy rule). The if_statement2basicblock rule, is an example of rule to deal with a control statement.

The right side of Figure 8 shows the *flow graph* concept used by the transformation template as target domain. It represents a directed graph, where the nodes are basic blocks and the edges are jumps to other basic blocks.

```
1  helper context Imp!Statement def: isLeader : Boolean =
2    if self.isControl then
3      true
4    else
5      let isTarget : Boolean =
6        Imp!GoToStatement.allInstances()->any(g | g.target = self.gotoLabel() )
7      in
8      if isTarget.oclIsUndefined() then
9          self.previousIsGoto
```

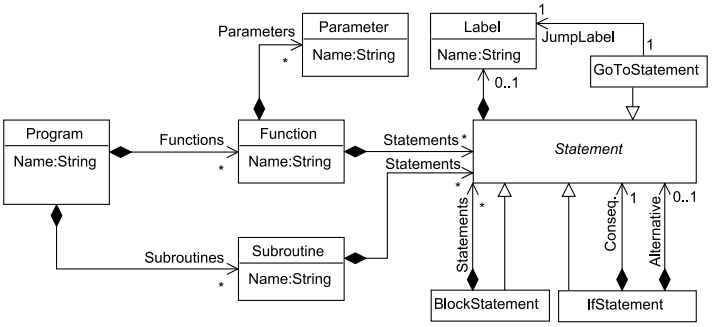


Figure 9 – Excerpt of the meta-model for NQC.

```
10      else
11          true
12      endif
13    endif
14  ;
15
16  rule statement2basicblock {
17    from stm : Imp!Statement (
18      (not stm.isControl) and stm.isLeader
19    )
20    using {
21      contents : Sequence(Imp!Statement) = stm.getBlock()
22    }
23    to block : BBL!BasicBlock (
24      edges <−
25        if contents−>last().oclIsKindOf(Imp!UnconditionalGoto) then
26          Sequence { thisModule.goto_statement2edge(contents−>last()) }
27        else
28          ...
29       end
30    )
31  }
32
33  rule if_statement2basicblock {
34    from stm : Imp!IfStatement
35    to block : BBL!BasicBlock (
36      edges <− Sequence {branch1, branch2},
37      identifier <− stm.toInfo
38    ),
39    branch1 : BBL!DirectedEdge ( ... ),
40    branch2 : BBL!DirectedEdge ( ... )
41  }
```

Listing 3 – Excerpt of the generic transformation to compute flow graphs.

In order to assess to what extent our approach is flexible enough to adapt meta-models not foreseen by the generic transformation developer we have applied the template for two different source meta-models of procedural languages, namely NQC (Not Quite C) and PL/SQL. These meta-models were defined by fellow researchers for projects independent from us [CM10, vAvdBE10].

First, we demonstrate our approach showing the binding for the NQC meta-model, and the binding to PL/SQL is discussed afterwards. Figure 9 shows an excerpt of the NQC meta-model, where a program is composed of Functions and Subroutines, which are in turn composed of Statements. In this case we are only interested in control statements such as IfStatement and GoToStatement.

There are several mismatches between the source concept and the NQC meta-model. Listing 4 shows the binding that solves these mismatches, namely:

- *Renamings.* For instance, Goto is mapped to GoToStatement (line 2), and the target reference of Goto is mapped to JumpLabel of GoToStatement (line 9).
- *Class with binding cardinality >1.* This is the case of Procedure that is bound to both Function and Subroutine (line 3).
- *Association to intermediate class association.* Statement in the concept and the meta-model can be mapped naturally, except when the statement is a BlockStatement, which from the partitioning algorithm point of view behaves as an association represented with an intermediate class. To tackle this issue, we use a binding adapter to calculate the collection resulting from navigating through the intermediate object (lines 13-15).
- *Monovalued association to multivalued association.* The IfStatement concept class has two multivalued associations, ifStms and elseStms, while the counterparts in the meta-model class are monovalued (Consequence and Alternative). In order to solve this heterogeneity, we define a binding adapter for each association (lines 17 and 19-21). These adapters make use of helpers factoring common code, which are also specified with our concrete syntax (flattened helpers in lines 23-24).

```
1  binding NQC for Imp {
2    class Goto to GoToStatement
3    class Procedure to Function, Subroutine
4
5    -- These bindings are optional (same name)
6    class Statement to Statement
7    class IfStatement to IfStatement
8
9    feature Goto.target is JumpLabel
10
11   feature Procedure.name is Name
12
13   feature Procedure.statements = self.Statements->collect(s |
14     if s.oclIsKindOf(BlockStatement) then s.Statements
15     else s endif )->flatten();
16
17   feature IfStatement.ifStms = self.Consequence.flattened;
18
19   feature IfStatement.elseStms =
20       if self.Alternative.oclIsUndefined() then Sequence { }
21       else self.Alternative.flattened endif;
22
23   helper Statement.flattened : Sequence(Statement) = Sequence { self };
24   helper BlockStatement.flattened : Sequence(Statement) = self.Statements;
25
26   -- Overriding toInfo hole to annotate the flow graph
27   helper IntegerConstant.toInfo : String = self.Value.toString()
28   helper AssignmentStatement.toInfo : String = self.Variable.toInfo + ' = ' + self.Expression.toInfo
29   ...
30 }
```

Listing 4 – Binding between the source concept and the NQC meta-model.

If we only bind the source concept, the execution of the adapted transformation yields a model conforming to the target concept (remember that a concept is implemented as regular meta-model).

If we would like to visualize the obtained basic blocks model, a model-to-text transformation can be built to generate Graphviz visualizations from the basic blocks model. However, this code generator must be built specifically for this purpose, becoming hard to reuse. Instead, we have created a Graphviz code generator whose inputs are models conforming to the meta-model shown in Figure 10. This meta-model

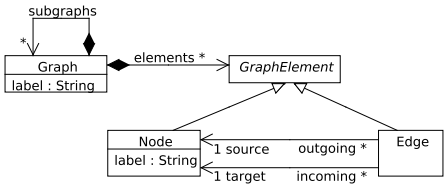


Figure 10 – Simple meta-model for Graphviz specifications

represents a simple Graphviz specification. It is reusable as far as one can map a graph-like model intended to be visualized to this meta-model, and then the code generator becomes reusable.

In this way, to reuse the Graphviz code generator we have to bind the target concept of the flow graph transformation to the Graphviz meta-model, as it is shown in Listing 5. The main problem to perform this binding correctly is the fact that, in the concept, there is a containment relationship between BasicBlock and Edge, while in the meta-model an Edge must be aggregated in the elements relationship. This mismatch cannot be addressed, since our binding mechanism is less flexible for target concepts than for source concepts. In order to fix this issue, ATL superimposition [WSD10] can be used to extend the adapted transformation so that the elements relationship is properly filled. In general, this strategy can be used to extend an adapted transformation when our binding is not powerful enough.

```
1  binding Graphviz for BBL {
2    class Routine to Graph
3    class BasicBlock to Node
4    class Edge to Edge
5
6    feature Routine.blocks is elements
7    feature Routine.name is name
8
9    feature BasicBlock.next is outgoing
10   feature BasicBlock.identifier is label
11
12   feature Edge.source is source
13 }
```

Listing 5 – Binding between the target concept and the Graphviz meta-model.

Figure 11 shows the result of applying the generic transformation to a NQC program that computes the dot product of two vectors (left part). The input model of the transformation is a regular EMF model (center), and the result is the corresponding flow graph (right part).

The second example is taken from a reverse engineering project of PL/SQL code. The meta-model of this language, shown in Figure 12, contains similar control constructs to the concept (e.g., *if* and *while*). However, there are some mismatches that must be addressed as well. Listing 6 shows the binding for this language, which is explained next.

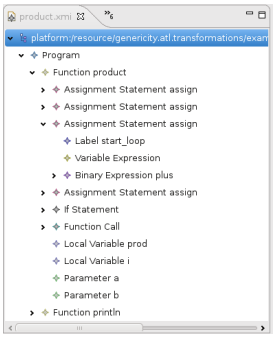
- First of all, even though PL/SQL supports the *goto* statement, the developers of the meta-models did not implement it. For this reason, we let the Goto class in the concept unbound, which is indicated by using the keyword NONE (line 2). In this case, our instantiation mechanism rewrites the template so as to ignore those parts dealing with *gotos*.

- Secondly, PL/SQL considers two kinds of loops: LoopStatement and ForStatement.

```
void product(int a[], int b[]) {
   int prod = 0;
   int i = 0;

start_loop:
   prod = prod + a[i] * b[i];
   i = i + 1;
   if ( i < 20 )
      goto start_loop;

   println(prod);
}
```



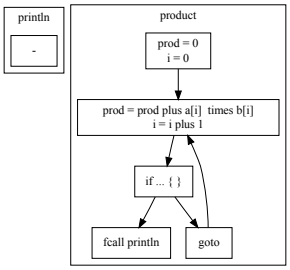


Figure 11 – Result of applying the generic transformation to a NQC program that computes the dot product of two vectors.

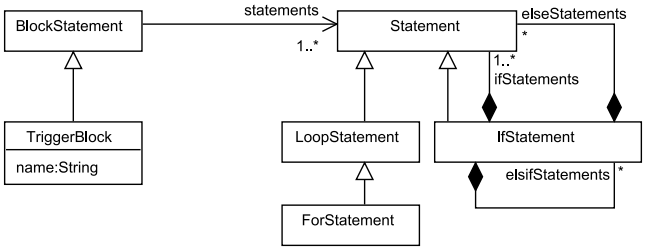


Figure 12 – Excerpt of the meta-model for PL/SQL.

Since ForStatement is a specialization of LoopStatement, we only bind the latter (line 7).

- Finally, in PL/SQL there is the notion of *elsif* to include more than one alternative branch in an *if* statement. However, our concept for procedural languages does not include such a construct, as the concept class IfStatement only considers one alternative branch through reference ifStmts. Thus, our binding does not consider the elsifStatements feature that is present in PL/SQL, and the transformation instantiated from our template produces a flow graph that ignores those branches. By including a case statement with cardinality 0..1 in our concept, and a new rule for its handling in the transformation template, we can specify a complete binding for PL/SQL. Thus, the construction of concepts and templates sometimes becomes an incremental procedure.

```
1  binding PlSql for Imp {
2    class Goto to NONE
3
4    class Procedure to TriggerBlock
5    class Statement to Statement
6    class IfStatement to IfStatement
7    class WhileStatement to LoopStatement
8
9    feature Procedure.statements = self.statements->collect(s |
10     if s.oclIsKindOf(BlockStatement) then s.statements
11     else s endif )->flatten();
12
13   feature IfStatement.ifStms = self.ifStatements
14   feature IfStatement.elseStms = self.elseStatements
15
16   -- Overriding toInfo hole to annotate the flow graph
```

```
17    helper AssignmentStatement.toInfo : String = self.receptor.toInfo + ' = ' + self.variable.toInfo
18    ...
19  }
```

Listing 6 – Binding between the source concept and the PL/SQL meta-model.

Altogether, we have used meta-models designed by two different developer teams, each one of them taking their own design decisions, independently from the ones we took when designing the concept. We would like to remark that both meta-models have been used in real projects, so the heterogeneities that have appeared are likely to appear in other cases as well. Our binding adapters and cardinalities in concept elements proved enough to overcome the heterogeneities, adapting the concept to unforeseen third-party meta-models.

Regarding the reusability gain, computing the flow graph of an imperative program is a non-trivial transformation, and therefore we do not want to implement it for each possible procedural language. In this case, the transformation template consists of 145 lines of code[1], some of them complex navigation code, while the bindings for NQC and PL/SQL have 40 and 46 lines of code respectively of a simpler specification (e.g., *toInfo* operations and simple mappings). Therefore, the cost of writing the binding models was much less than the cost of reimplementing the transformation from scratch for each different meta-model. Moreover, when reusing a transformation template, the user does not need to become familiar with the intricate details of its implementation, but only needs to understand and bind the concept (expressing the requirements for the transformation) with his particular meta-model. Besides, it is easier to write the transformation template against the concept than against an arbitrary meta-model, as the concept is not cluttered with details irrelevant for the purposes of the transformation.

In summary, in this case study we were able to follow the motto *"write once, reuse everywhere"* by designing a suitable concept and defining the transformation over the concept.

7 Related work

Generic programming is a programming paradigm found in many languages like C++, Haskell or Java [GJL+03]. C++ supports generic programming by the template system. An operation can be made generic by defining a template function that contains a set of type parameters. However, the requirements of a type parameter are not explicitly expressed. Concepts were proposed to overcome this limitation [GJS+06], but they have not been eventually included in C++0x, the last revision of C++ [Str09]. In Haskell, the requirements of a generic operation (i.e., a polymorphic function) are expressed through *type classes* [Jon02]. A type can be made an instance of a given type class in order to make it compatible with it. Thus, implementing a type class is the Haskell equivalent to our binding mechanism. In Scala, requirements on a type parameter can be expressed with a *trait*. It can be implemented in a so-called *object*, and automatically selected for instantiation using the *implicit* mechanism [OMO10]. Regarding the expressive power of these approaches, all of them are comparable to our basic binding mechanism plus adapters. However, they require 1-1 bindings, that is, every type parameter has to be mapped exactly once, while we permit specifying cardinality in concept elements.

[1] Removing comments and blank lines.

Meta-model concepts were first proposed in [dLG10] with an application to the definition of generic in-place transformations using EOL [KPP06]. The architecture in [dLG10] uses an interpreted approach for the instantiation of templates which does not generate new transformations, but it uses the binding to resolve the concrete types at run-time. The work in [RGdL$^+$11] also uses a concept-based interpreted approach to define generic model management operations (not just in-place transformations) using the Epsilon languages, where some concept elements can be decorated with usages (creation or deletion) according to the operations performed by the generic operation. In the present paper, we use a compiled approach on top of ATL, where a HOT creates a specific transformation according to the binding. Moreover, the binding function in [dLG10, RGdL$^+$11] is 1-to-1, whereas here we propose a more flexible structural binding (e.g., abstract classes in the concept need not be bound) and two mechanisms to enhance flexibility: adapters and replication of concept elements. Finally, our approach does not require the use of concept usage annotations, but we have defined constraints for read-enabled and write-enabled domains. In [dLG11], in the context of graph transformation, we developed an algebraic formalization of concepts and meta-models, which enabled the expression of bindings as morphisms. However, that formalization accounted for 1-to-1 bindings, not supporting adapters and cardinalities as we propose here.

The term transformation template has also been used in previous works, although with a different intention. For instance, in [KG07] the authors build transformation templates for a family of languages defined by a unique meta-model. Variations in this meta-model induce modifications in the template. However, it is not possible to apply a template to unrelated meta-models as we do here. In [KKCS10], the authors present the graphical MOLA template language, which can be used to specify generic transformations in the MOLA transformation language. Its purpose is to provide a concrete syntax to create transformations, in contrast to HOTs which usually work at the abstract syntax level and therefore are more complex to specify. Our templates also work at the concrete syntax level, in particular we use ATL syntax in this paper.

Other approaches to reusability are not based on concepts. For instance, in [SMM$^+$10], reuse is achieved by adapting the meta-models to which an existing transformation is to be applied. The aim of adapting the meta-model is to make it a subtype of the expected input meta-model of the transformation [SJ07], so that the transformation can be applied without changing it. In contrast, our approach is less intrusive because we do not need to modify the meta-models, which sometimes can be unfeasible. Moreover, once a template is instantiated, we can extend the generated transformation with rules using concrete types. Our binding is also similar to the notion of model subtyping [SJ07], as one can see a concept as a supertype of the bound meta-model; however, our cardinality and adapters makes this relation more flexible than pure subtyping.

A few transformation languages support the definition of parameterised rules. For instance, in [ALS08], transformation rules can define *String* parameters with the name of classes, attributes and associations which are resolved at run-time. In [VP04], the type of the objects in a rule can be variables which are also resolved at run-time. In addition, in this latter approach one can build meta-transformations to generate first-order rules where the variables are substituted by concrete types of a specific language, thus obtaining more efficient rules. In our case, we also generate specialized transformations from our templates, however this is done by a transformation-independent HOT that can be used for any generic template and

binding. The MOLA Template language [KKCS10] permits the definition of generic rules where some (or all) types can be variables, which are concretised at generation time, yielding a regular MOLA transformation. However, none of these works [ALS08, KKCS10, VP04] provide mechanisms – similar to our concepts – to express the requirements that the variable types should fulfil, nor correctness rules to ensure that a particular transformation instance is valid.

Another approach to reuse are the mapping operators (MOps) [WRK+10]. These are similar to our adapters, but oriented to the declarative construction of transformations by composing transformation primitives. Reusable transformations must be completely developed using MOPs, while we permit using a regular transformation language. The same authors present in [WKK+10] a categorization of common heterogeneities, which we solve through adapters and cardinality in concepts.

The formal language Maude also supports genericity and generic bindings (so called parameterized views) [DM00], similar to our proposal of generic adapters. However, Maude is a formal specification language enriched with rewriting logic constructs, and not properly a model-to-model transformation language.

Finally, adapting a transformation via a HOT as a response to meta-model changes has already been proposed in the field of transformation-metamodel coevolution [EMMC10, REM+10]. In fact, our adaptation mechanism could be used for the purpose of migration, by generating binding specifications describing how a version of the same meta-model has evolved.

8 Conclusions and Future Work

In this paper we have brought elements of generic programming to model-to-model transformations in order to promote reusability. In our approach, it is possible to define transformation templates that use generic types defined on a concept. Concepts can be bound to a range of specific meta-models satisfying the concept requirements. In this way, transformation templates can be instantiated for different meta-models and applied to the meta-model instances. We have proposed two mechanisms to provide flexibility to the binding function and resolve heterogeneities between concepts and meta-models: cardinality annotations and binding adapters. We have implemented this approach atop ATL, and illustrated its use through a non-trivial example. The tool is available at `http://sanchezcuadrado.es/projects/genericity`.

In the future, we plan to extend our categorization of binding adapters, and to implement libraries of reusable adapters and reusable transformation templates. We are also improving the HOT that instantiates the transformation template to automate the resolution of certain heterogeneities, like for example the binding of multi-valued features to mono-valued ones. We would also like to apply our approach to other transformation languages and explore composition mechanisms for generic transformations. It would also be interesting to investigate the degree in which handmade, customized HOTs [TJF+09] can be represented using a template mechanism and a binding model, as we have presented in this paper. Finally, we are also aiming at extending the algebraic formalization we started to develop in [dLG11] to take into account adapters and cardinalities in concepts.

Acknowledgements. This work was funded by the Spanish Ministry of Economy and Competitivity (project "Go Lite" TIN2011-24139) and the R&D programme of the Madrid Region (project "e-Madrid" S2009/TIC-1650). We also thank the referees for their valuable comments.

References

[ALS08] Carsten Amelunxen, Elodie Legros, and Andy Schürr. Generic and reflective graph transformations for the checking and enforcement of modeling guidelines. In *VLHCC'08*, pages 211–218. IEEE, 2008. doi:10.1109/VLHCC.2008.4639088.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[BGdL10] Paolo Bottoni, Esther Guerra, and Juan de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *IST*, 52(8):821–844, 2010. doi:10.1016/j.infsof.2010.03.005.

[BJP05] J. Bézivin, F. Jouault, and J. Palies. Towards model transformation design patterns. In *EWMT'05*, 2005.

[BL93] Thomas Ball and James R. Larus. Branch prediction for free. *SIGPLAN*, 28:300–313, 1993. doi:10.1145/173262.155119.

[CGdL11] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Generic model transformations: *Write Once, Reuse Everywhere*. In *ICMT'11*, volume 6707 of *LNCS*, pages 62–77. Springer, 2011. doi:10.1007/978-3-642-21732-6_5.

[CM10] Javier Cánovas and Jesús García Molina. An architecture-driven modernization tool for calculating metrics. *IEEE Software*, 27(4):37–43, 2010. doi:10.1109/MS.2010.61.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–523, 1985. doi:10.1145/6041.6042.

[dLG10] Juan de Lara and Esther Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MoDELS'10*, volume 6394 of *LNCS*, pages 16–30. Springer, 2010. doi:10.1007/978-3-642-16145-2_2.

[dLG11] Juan de Lara and Esther Guerra. Reusable graph transformation templates. In *AGTIVE'11*, 2011.

[DM00] Francisco Durán and José Meseguer. Parameterized Theories and Views in Full Maude 2.0. *ENTCS*, 36, 2000. doi:10.1016/S1571-0661(05)80136-7.

[EMMC10] Anne Etien, David Mendez, Alexis Muller, and Rubby Casallas. Towards transformation migration after metamodel evolution. In *International Workshop on Models and Evolution (ME'10)*, 2010.

[FP09] Steven Feuerstein and Bill Pribyl. *Oracle PL/SQL Programming, Fifth Edition*. O'Reilly Media, 2009.

[Gen] Generic model transformations. http://sanchezcuadrado.es/projects/genericity.

[GJL+03] Ronald García, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic

programming. *SIGPLAN*, 38(11):115–134, 2003. doi:10.1145/949343.949317.

[GJS+06] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. *SIGPLAN Not.*, 41(10):291–310, 2006. doi:10.1145/1167515.1167499.

[JABK08] Frederic Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. doi:10.1016/j.scico.2007.08.002.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *GPCE'06*, pages 249–254. ACM, 2006. doi:10.1145/1173706.1173744.

[Jon02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, September 2002. Available from: http://haskell.org/definition/haskell98-report.pdf.

[KG07] Amogh Kavimandan and Aniruddha Gokhale. A parameterized model transformations approach for automating middleware QoS configurations in distributed real-time and embedded systems. In *WRASQ'07*, 2007. doi:10.1145/1314483.1314487.

[KKCS10] Elina Kalnina, Audris Kalnins, Edgars Celms, and Agris Sostaks. Graphical template language synthesis. In *SLE'09*, volume 5969 of *LNCS*, pages 244–253. Springer, 2010. doi:10.1007/978-3-642-12107-4_18.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006. doi:10.1007/11787044_11.

[LS06] Michael Lawley and Jim Steel. Practical declarative model transformation with Tefkat. In *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, MoDELS'05, pages 139–150, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11663430_15.

[OMG] OMG. UML 2.3 specification. http://www.omg.org/spec/UML/2.3/.

[OMG05] OMG. Final adopted specification for MOF 2.0 Query/View/Transformation, 2005.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *SIGPLAN Not.*, 45(10):341–360, October 2010. doi:10.1145/1932682.1869489.

[REM+10] Louis Rose, Anne Etien, David Mendez, Dimitrios Kolovos, Richard Paige, and Fiona Polack. Comparing model-metamodel and transformation-metamodel co-evolution. In *International Workshop on Models and Evolution (ME'10)*, 2010.

[RGdL+11] Louis Rose, Esther Guerra, Juan de Lara, Anne Etien, Dimitris Kolovos, and Richard Paige. Genericity for model management operations. *Software and System Modeling*, In press, 2011. doi:10.1007/s10270-011-0203-2.

[SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.

[SJ07] Jim Steel and Jean-Marc Jézéquel. On model typing. *Software and System Modeling*, 6(4):401–413, 2007. doi:10.1007/s10270-006-0036-6.

[SM09] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison Wesley, 2009.

[SMM+10] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jézéquel. Reusable model transformations. *Software and System Modeling*, In press, 2010. doi:10.1007/s10270-010-0181-9.

[Str09] Bjarne Stroustrup. The C++0x remove concepts decision. *Dr.Dobbs*, 2009. http://www.ddj.com/cpp/218600111.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the use of higher-order model transformations. In *ECMDA-FA'09*, volume 5562 of *LNCS*, pages 18–33. Springer, 2009. Available from: http://dx.doi.org/10.1007/978-3-642-02674-4_3.

[vAvdBE10] Marcel van Amstel, Mark van den Brand, and Luc Engelen. An exercise in iterative domain-specific language design. In *IWPSE-EVOL'10*, pages 48–57. ACM, 2010. doi:10.1145/1862372.1862386.

[VP04] Dániel Varró and András Pataricza. Generic and meta-transformations for model transformation engineering. In *UML'04*, volume 3273 of *LNCS*, pages 290–304. Springer, 2004. doi:10.1007/978-3-540-30187-5_21.

[VS06] Markus Völter and Thomas Stahl. *Model-driven software development*. Wiley, 2006.

[WKK+10] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Towards an expressivity benchmark for mappings based on a systematic classification of heterogeneities. In *MDI'10*, pages 32–41. ACM, 2010. doi:10.1145/1866272.1866278.

[WRK+10] M. Wimmer, W. Retschitzegger, G. Kappel, J. Schoenboeck, A. Kusel, and W. Schwinger. Plug & play model transformations: A DSL for resolving structural metamodel heterogeneities. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, DSM '10, pages 7:1–7:6, New York, NY, USA, 2010. ACM. doi:10.1145/2060329.2060348.

[WSD10] Dennis Wagelaar, Ragnhild Van Der Straeten, and Dirk Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *Software and System Modeling*, 9(3):285–309, 2010.