Generic Model Transformations: *Write Once, Reuse Everywhere*

Jesús Sánchez Cuadrado[1], Esther Guerra[2], and Juan de Lara[2]

[1] Universidad de Murcia (Spain)
jesusc@um.es
[2] Universidad Autónoma de Madrid (Spain)
{Esther.Guerra, Juan.deLara}@uam.es

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 defined over *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.

1 Introduction

Model Driven Engineering (MDE) proposes the use of models as the key assets in the development, and hence all sorts of model modifications are needed. In this way, model transformations become one of the basic building blocks of MDE.

Even though MDE is being successfully applied 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 transformations reusable, so that we can apply them to different contexts (i.e. with different meta-models). This would enable the creation of transformation patterns and idioms [3], as well as libraries of transformations addressing recurrent transformation problems. Some examples of 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. 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. In particular, we propose defining model transformation templates over concepts [6, 10, 17]. 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 in a transformation. Thus, when then concept is bound to a specific meta-model satisfying the concept requirements, the transformation becomes applicable to this meta-model.

In [6] we proposed concepts as a mechanism to add genericity to models, meta-models and in-place transformations. However, we only allowed a restricted kind of binding between the concepts and the meta-models consisting in 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 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 [11] where the adaptation of the transformation templates is realized by a higher-order transformation (HOT). Nonetheless, our approach is general and therefore applicable to other transformation languages.

Paper Organization. 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 flexibility to our approach by providing multiple cardinality for our concepts and adapters for the bindings. Section 5 outlines our exemplary implementation on top of ATL and Section 6 presents a case study. Section 7 compares with related work and Section 8 concludes.

2 Genericity in model transformation

Genericity is a programming paradigm found in many languages like C++, Haskell, Eiffel or Java [8]. 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 [10, 17].

Genericity is realized through 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* [10], 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 of their type parameters.

Based on these ideas, we have defined the approach for generic model transformations that is outlined in Fig. 1. Similar to programming templates, we build

generic model transformation templates. These are transformations in which the source or the target domain (or both) is not tied to a specific meta-model, but contains *variable* types. The requirements for the variables types (needed properties, associations, etc.) are specified through a concept. A concept has the form of a meta-model as well, but its enclosing elements are interpreted as variables that need to be bound to elements of some concrete meta-model.

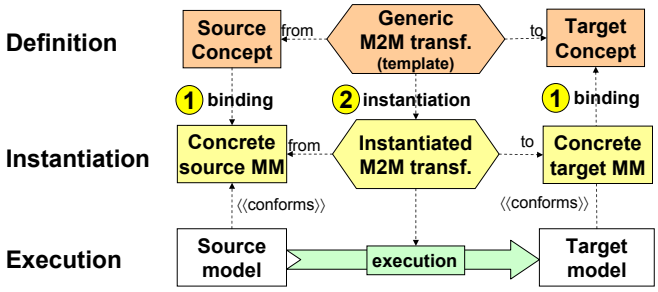

Fig. 1. Working scheme of our approach. In this case both the source and target domains are concept meta-models.

Once bindings between concepts and concrete meta-models are established (step 1 in Fig. 1), 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 could be a concrete meta-model.

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: multiple cardinality (or variability of concept elements) and adaptation. As we will see, the more sophisticated the binding is, the more complex the instantiation of the transformation becomes. In this paper, the instantiation mechanism for template transformations is implemented 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 (or in general, of a generic model management operation [6]). 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 generic transformation template.

As an example, the upper part of Fig. 2 shows the definition of a generic transformation that populates 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, generates a Java class from each OO class, and a Java field from each attribute.

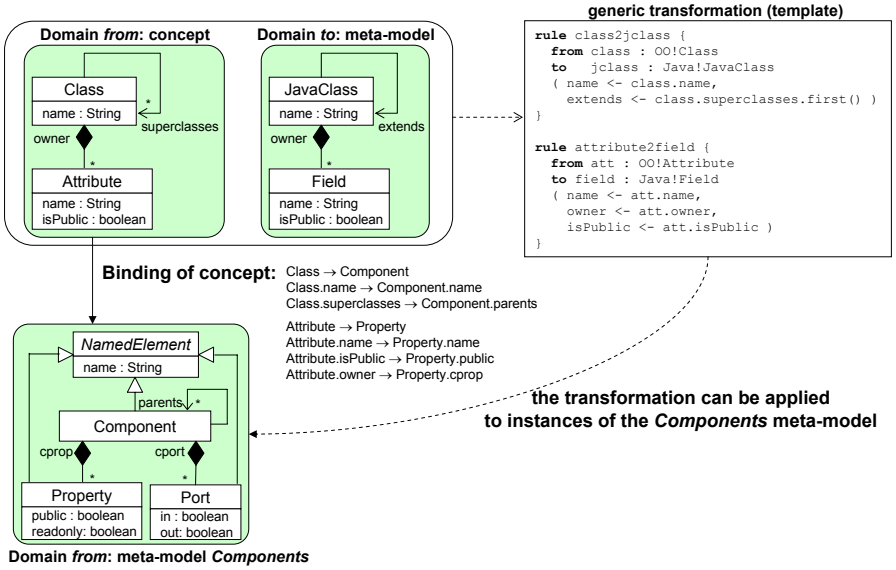

Fig. 2. Transformation template from a concept for OO systems into a fixed Java meta-model, and binding of the concept to a particular meta-model for components.

In order to execute a generic transformation we have to map or bind the concepts involved in the template to specific meta-models. If one represents algebraically concepts and meta-models as attributed type graphs with inheritance [5], then, in the simplest scenario, the binding is a morphism (a function) that takes into consideration the semantics of inheritance (so-called clan-morphisms in [5]). In the following we just provide an intuition of this binding function and purposely refrain from a formalization.

In the simplest case, 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. The binding also imposes some additional conditions to ensure the compatibility of the mapped associations and attributes, so that the operations performed in the concept can be performed in the bound meta-model types. In particular:

- the source and target classes of an association must be mapped to the source and target classes of the bound association (or to some of their subclasses).
- Attributes must be mapped to other attributes of the same type, or of a subtype. Moreover, the container classes of two mapped attributes must also be bound. There is an exception though: it is allowed to map an attribute to an inherited attribute (i.e. an attribute defined in a superclass). For instance, in Fig. 2, attribute name of Class is bound to attribute name of NamedElement, although Class is not mapped to NamedElement. Nonetheless this is allowed because Class is bound to Component, of which NamedElement is a superclass.
- if a concept represents the target of a transformation, meaning that its instances (or more precisely the instances of the bound meta-models) are being populated by the transformation, then it is not possible to map an association in the concept with an association with a lower upper bound or higher lower bound.

The binding should be a function from the set of classes, associations and attributes. 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 do not lead to conflicting rules. 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, Fig. 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 binding maps classes Class and Attribute in the concept to classes Component and Property in the meta-model, associations superclasses and owner to parents and cprop, 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. 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-model(s). The first possibility is to encode a HOT that takes as input the template, the bound meta-model(s) and the binding(s), and produces a transformation that is directly applicable to the bound meta-model(s). The HOT replaces in the template each class, association and attribute declared in the concept by elements in the concrete meta-model, as specified by the binding. 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 [6].

4 Adding flexibility to concepts and bindings

The presented binding mechanism enables a certain degree of reuse, but it lacks flexibility as the meta-model must embed the concept, that is, the concept and the bound part of the meta-model must be structurally equal.

For example, in Fig. 2 we may also want to treat Ports as Attributes 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 [19].

4.1 Cardinality of concept elements

Our previous binding definition requires each element in the concept to be mapped to exactly one element in the bound 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 classes and associations 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 generic transformation 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.

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 [4], 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. Fig. 3 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 a particular meta-model to the concept. For instance, in order to map Attribute to both Property and Port in the components meta-model shown in Fig. 2, we must select two replicas. Then, one can perform a 1-to-1 binding from the corresponding expanded concept to the meta-model.

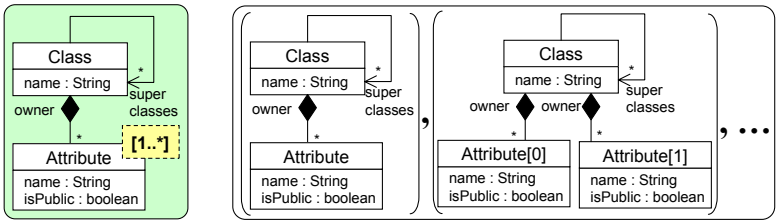

Fig. 3. Annotating a concept with cardinality (left). Set of its expansions (right).

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 the cardinality is defined for an association, then the instructions assigning a value to the association should be replicated for each specified binding. For instance, if the target domain in Fig. 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.

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 Fig. 2 could be implemented with an intermediate class, as in the case of the UML meta-model [14], 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 resorting to subtyping relations between concepts [16]. 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. Fig. 4 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 can be extended, which sometimes can be difficult or undesired. Moreover, not all transformation languages support rule extension.

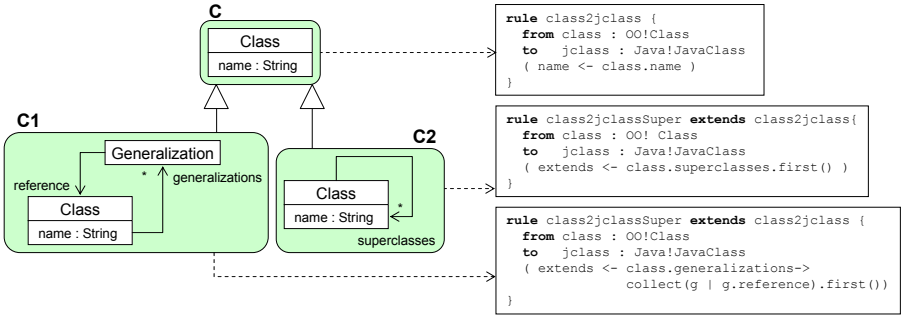

Fig. 4. Binding concepts to concepts, seen as a subtyping relation between concepts.

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.

Fig. 5 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 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.

A side 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

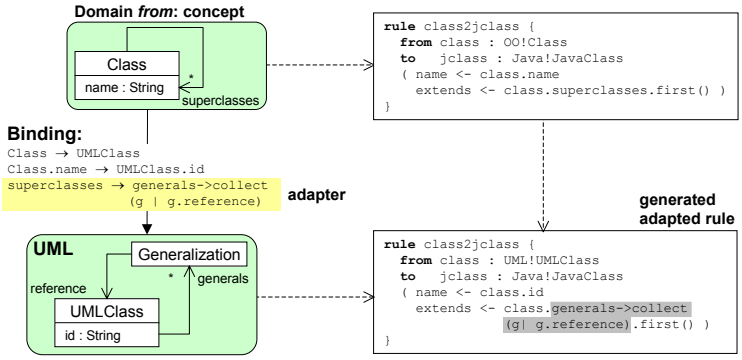

Fig. 5. A binding adapter (left). Semantics of adapter as template modification (right).

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 Fig. 5.

5 Implementing genericity over ATL

We have implemented a prototype, available at [9], to support our approach to generic model transformations. It currently targets ATL.

In our tool, a concept is defined as an Ecore meta-model, 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 to describe such bindings, and use OCL to define adapters. As an example, Listing 1 shows the binding presented in Fig. 2 expressed with our concrete syntax. Class is mapped to Component (line 2), and Attribute is mapped to both Property and Port (line 3). 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). Our tool also allows some mappings to be implicit. For instance, it is not needed to map Class.name to Component.name. Finally, the None keyword allows a concept element to be unbound whenever its minimum cardinality is 0.

```
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.

Templates are written in ATL. 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. 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.* Currently, it is restricted to rules with one *from* element. 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.
  - *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.* It is restricted to rules with one *from* element. 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 adapter*. 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 name of concept features, and the ATL engine selects the appropriate attribute helper dynamically.

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 before.
- *Feature with cardinality 0*. The ATL binding is removed.

Please note that binding adapters are only possible for source concepts.

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 analysis [2].

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 choose to build just a unique transformation template defined over such a concept. In this way, the template will be applicable to every procedural language once we bind their meta-model to the concept.

Fig. 6 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.

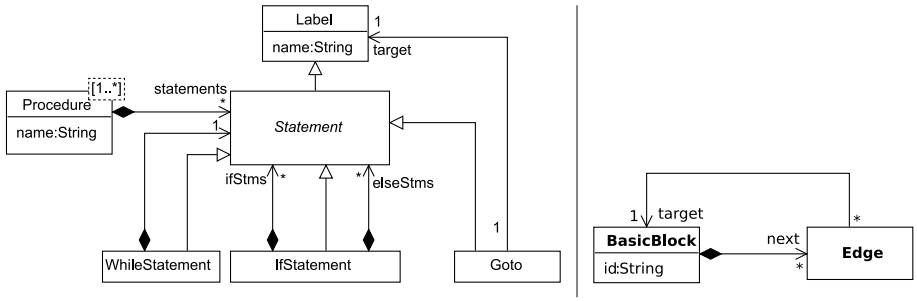

Fig. 6. Concept for imperative languages (left). Meta-model for control flows (right).

Our transformation template implements a variation of the algorithm proposed in [1], 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 space limitations, but it is worth noting that this is not a straightforward ATL transformation, comprising 7 rules and 13 helpers.

The right side of Fig. 6 shows the *flow graph* meta-model 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.

We have instantiated our template to transform programs written in NQC (Not Quite C), a C-like language used to program Lego Mindstorms. We have used a meta-model defined by a third party [18] in order to assess to what extent our approach is flexible enough to adapt meta-models not foreseen by the generic transformation developer. Fig. 7 shows an excerpt of the NQC meta-model.

There are several mismatches between the source concept and the NQC meta-model. Listing 3 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 5).
- *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. We use a binding adapter to tackle this issue (lines 9-11).

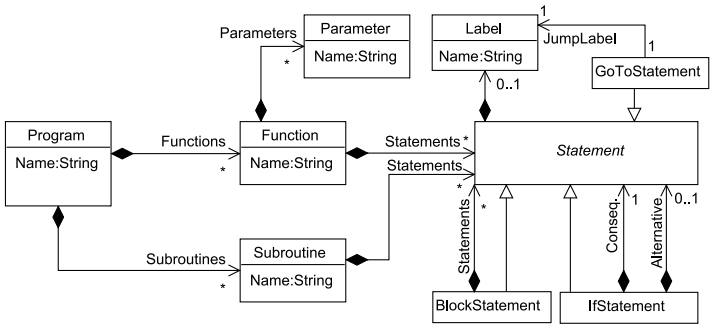

Fig. 7. Excerpt of the meta-model for NQC.

– *Monovalued association to multivalued association.* The IfStatement concept class has two multivalued associations, ifStms and elseStms, while the counterparts in the concrete class are monovalued (Consequence and Alternative). Moreover, the else clause is optional. In order to solve this heterogeneity, we define a binding adapter for each association (lines 13 and 15-17). These adapters make use of helpers factoring common code, which are also specified with our concrete syntax (flattened helpers in lines 19-20).

```
1  binding nqc {
2    class Goto to GoToStatement
3    class Procedure to Function, Subroutine
4
5    feature Goto.target      is JumpLabel
6
7    feature Procedure.name is Name
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.Consequence.flattened;
14
15   feature IfStatement.elseStms =
16       if   self.Alternative.oclIsUndefined() then Sequence { }
17       else self.Alternative.flattened endif;
18
19   helper      Statement.flattened : Sequence(Statement) = Sequence { self };
20   helper BlockStatement.flattened : Sequence(Statement) = self.Statements;
21    ...
22 }
```

Listing 3. Binding between the source concept and the NQC meta-model.

Altogether this case study shows the feasibility of our approach, but two issues are worth noting. First, 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. Here we were able to follow the motto *"write once, reuse everywhere"* by designing a suitable concept and defining the transformation over the concept. Second, our binding proved to be flexible enough to adapt the concept to an unforeseen third-party meta-model.

7 Related work

Meta-model concepts were first proposed in [6], with an application to the definition of generic in-place transformations using EOL [13]. The architecture in [6] 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. 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 [6] is 1-to-1, whereas here we propose two mechanisms to enhance flexibility: adapters and replication of concept elements.

The term transformation template has also been used in previous works, although with a different purpose. For instance, in [12] 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.

Other approaches to reusability are not based on concepts. For instance, in [15], 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 [16], 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. Nonetheless, our binding is similar to the notion of model subtyping [16], 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.

Another approach to reuse are the mapping operators (MOps) [19]. 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 [20] a categorization of common heterogeneities, which we solve through adapters and cardinality in concepts.

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 [9].

In the future, we plan to extend our categorization of binding adapters, and to implement libraries of reusable adapters and reusable transformation templates. We would also like to apply our approach to other transformation languages.

Acknowledgements. Work funded by the Spanish Ministry of Science (projects TIN2008-02081 and TIN2009-11555), and the R&D programme of the Madrid Region (project S2009 /TIC-1650).

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.
2. T. Ball and J. R. Larus. Branch prediction for free. *SIGPLAN*, 28:300–313, 1993.
3. J. Bézivin, F. Jouault, and J. Palies. Towards model transformation design patterns. In *EWMT'05*, 2005.
4. P. Bottoni, E. Guerra, and J. de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *IST*, 52(8):821–844, 2010.
5. J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Attributed graph transformation with node type inheritance. *TCS*, 376(3):139–163, 2007.
6. J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MoDELS'10*, volume 6394 of *LNCS*, pages 16–30. Springer, 2010.
7. F. Durán and J. Meseguer. Parameterized theories and views in full maude 2.0. *ENTCS*, 36, 2000.
8. R. García, J. Jarvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. *SIGPLAN*, 38(11):115–134, 2003.
9. Generic model transformations. `http://www.modelum.es/projects/genericity`.
10. D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. dos Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. *SIGPLAN Not.*, 41(10):291–310, 2006.
11. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. See also `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page`. Last accessed: Nov. 2010.
12. A. Kavimandan and A. Gokhale. A parameterized model transformations approach for automating middleware QoS configurations in distributed real-time and embedded systems. In *WRASQ'07*, 2007.
13. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
14. OMG. UML 2.3 specification. `http://www.omg.org/spec/UML/2.3/`.
15. S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. *Software and System Modeling*, In press, 2010.
16. J. Steel and J.-M. Jézéquel. On model typing. *SoSyM*, 6(4):401–413, 2007.
17. A. Stepanov and P. McJones. *Elements of Programming*. Addison Wesley, 2009.
18. M. van Amstel, M. van den Brand, and L. Engelen. An exercise in iterative domain-specific language design. In *IWPSE-EVOL'10*, pages 48–57. ACM, 2010.
19. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Plug & play model transformations - a DSL for resolving structural metamodel heterogeneities. In *DSM'10*. Online Publication, 2010.
20. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Towards an expressivity benchmark for mappings based on a systematic classification of heterogeneities. In *MDI'10*, pages 32–41. ACM, 2010.