Generic Model Refactorings*

Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel

INRIA Rennes – Bretagne Atlantique/IRISA, Université Rennes 1,
Triskell Team, Campus de Beaulieu, 35042 Rennes Cedex, France
{moha,vmahe,barais,jezequel}@irisa.fr

Abstract. Many modeling languages share some common concepts and principles. For example, Java, MOF, and UML share some aspects of the concepts of classes, methods, attributes, and inheritance. However, model transformations such as refactorings specified for a given language cannot be readily reused for another language because their related metamodels may be structurally different. Our aim is to enable a flexible reuse of model transformations across various metamodels. Thus, in this paper, we present an approach allowing the specification of generic model transformations, in particular refactorings, so that they can be applied to different metamodels. Our approach relies on two mechanisms: (1) an adaptation based mainly on the weaving of aspects; (2) the notion of model typing, an extension of object typing in the model-oriented context. We validated our approach by performing some experiments that consisted of specifying three well known refactorings (Encapsulate Field, Move Method, and Pull Up Method) and applying each of them onto three different metamodels (Java, MOF, and UML).

Keywords: Adaptation, Aspect Weaving, Genericity, Model Typing, Refactoring.

1 Introduction

Software reuse has been largely investigated in the last two decades by the software engineering community [3,23]. Basili *et al.* [2] have demonstrated the benefits of software reuse on the productivity and quality in object-oriented systems. In the domain of Model-Driven Engineering (MDE), which is often based on object-oriented metamodels, few works have been devoted to model-driven reuse [5]. For example, many modeling languages share some common concepts and principles: Java, MOF, and UML share some aspects of the concepts of classes, methods, attributes, and inheritance. However, a given model transformation such as the refactoring Pull Up Method specified for the UML metamodel might not be reused, for instance, for the Java metamodel because these metamodels are structurally different. Thus, the specification of model transformations are highly dependent on specific metamodels. Our aim is to

* This work was realized in the context of the MOVIDA project, funded by the ANR (French National Research Agency) CONVENTION N 2008 SEGI 011.

A. Schürr and B. Selic (Eds.): MODELS 2009, LNCS 5795, pp. 628–643, 2009.

enable a flexible reuse of such model transformations across various metamodels to enhance productivity and quality in the model-driven development.

In this paper, we present an approach to specify model transformations in a generic way, so that they can be applied to different metamodels. Our approach relies on two mechanisms: (1) an adaptation based mainly on the weaving of aspects; (2) the notion of model typing [31], an extension of object typing in the model-oriented context. We choose to illustrate and demonstrate our approach on well known model transformations, namely refactorings [10]. A refactoring is a particular transformation performed on the structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [10]. For example, the refactoring Pull Up Method consists of moving methods to the superclass if these methods have same signatures and/or results on subclasses [10]. We validated our approach by performing some experiments that consisted of specifying three well known refactorings (Encapsulate Field, Move Method, and Pull Up Method) and applying each of them onto three different metamodels (Java, MOF, and UML). The specification of refactorings has been performed with Kermeta, a meta-language for defining the structure and behavior of models [25].

This article is organized as follows. Section 2 provides an overview of our motivation. Section 3 introduces the executable metamodeling language, Kermeta, and highlights some of its new features including the notion of model typing. Section 4 presents our approach along with the Pull Up Method refactoring. Section 5 describes the experiments that we performed for the three refactorings (Encapsulate Field, Move Method, and Pull Up Method) on three different metamodels (Java, MOF, and UML). Section 6 surveys related work. Section 7 concludes and presents future work.

2 Motivation

Our motivation is to enable the specification of generic refactorings, so that they can be applied to different metamodels. In this section, we clearly state this motivation using the concrete example of the Pull Up Method refactoring on three different metamodels (Java, MOF, and UML).

2.1 The Pull Up Method Refactoring

The Pull Up Method refactoring consists of moving methods to the superclass when methods with identical signatures and/or results are located in sibling subclasses [10]. This refactoring aims to eliminate duplicate methods by centralizing common behavior in the superclass. A set of preconditions must be checked before applying the refactoring. For example, one of the preconditions to be checked consists of verifying that the method to be pulled up is not a constructor. Another precondition checks that the method does not override a method of the superclass with the same signature. A third precondition consists of verifying that methods in sibling subclasses have the same signatures and/or results.

The example of the Pull Up Method refactoring presented in [22] of a Local Area Network (LAN) application [15] and adapted in Figure 1 shows that the method `bill` located in the classes `PrintServer` and `Workstation` is pulled up to their superclass `Node`.

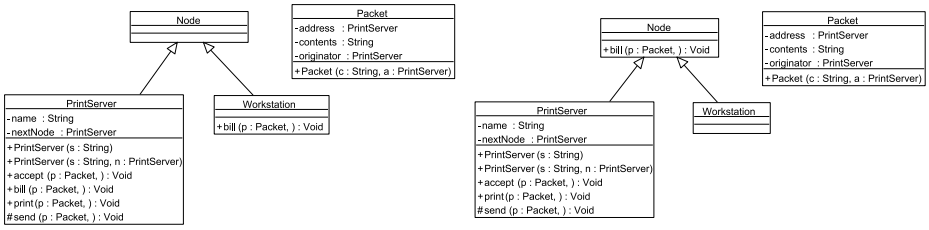

Fig. 1. Class Diagrams of the LAN Application Before and After the Pull Up Method Refactoring of the Method `bill`

2.2 Three Different Metamodels

We consider three different metamodels (Java, MOF, and UML), which support the definition of object-oriented structures (classes, methods, attributes, and inheritance). The Java metamodel described in [14] represents Java programs with some restrictions over the Java code. For example, inner classes, anonymous classes, and generic types are not modeled. As MOF metamodel, we consider the metamodel of Kermeta [25], which is an extension of MOF [27] with an imperative action language for specifying constraints and operational semantics of metamodels. The UML metamodel studied in this paper corresponds to the version 2.1.2 of the UML specification [29]. This Java metamodel is *one* possible representation of Java programs; there is no standard for such metamodel in contrast to UML and MOF metamodels.

We provide an excerpt of each of these metamodels in Figures 2, 3, and 4. These metamodels share some commonalities, such as the concepts of classes, methods, attributes, parameters, and inheritance (highlighted in grey in the figures). These concepts are necessary for the specification of refactorings, and in particular for the Pull Up Method refactoring. However, they are represented differently from one metamodel to another as detailed in the next paragraph.

2.3 Problems

We list here some of the problems encountered when trying to specify one common Pull Up Method refactoring for all three metamodels:

– **The metamodel elements** (such as classes, methods, attributes, and references) **may have different names**.

For example, the concept of attribute is named `Property` in the MOF and UML metamodels whereas in the Java metamodel, it is named `Variable`.

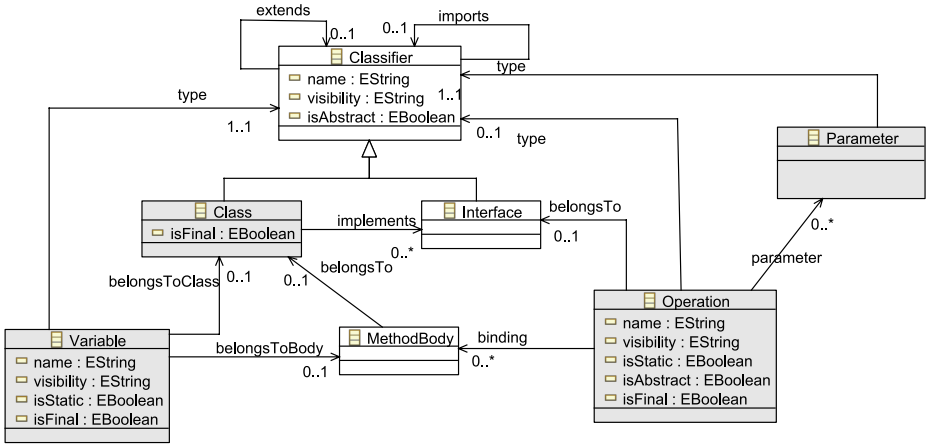

Fig. 2. Subset of the Java Metamodel

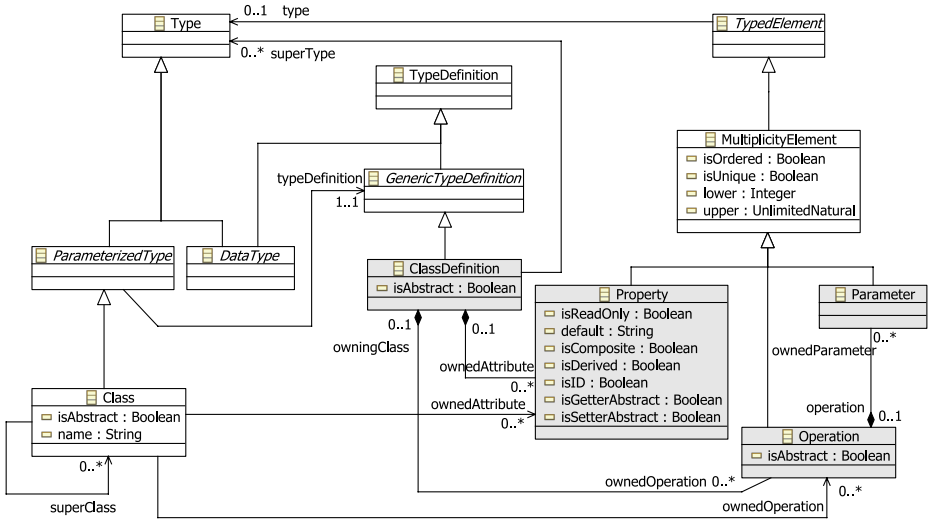

Fig. 3. Subset of the MOF Metamodel

– **The types of elements may be different**.

  For example, in the UML metamodel, the attribute `visibility` of `Operation` is an enumeration of type `VisibilityKind` whereas the same attribute in the Java metamodel is of type `String`.

– There may be **additional or missing elements** in a given metamodel compared to another.

  For example, `Class` in the UML metamodel and `ClassDefinition` in the MOF metamodel have several superclasses whereas `Class` in the Java

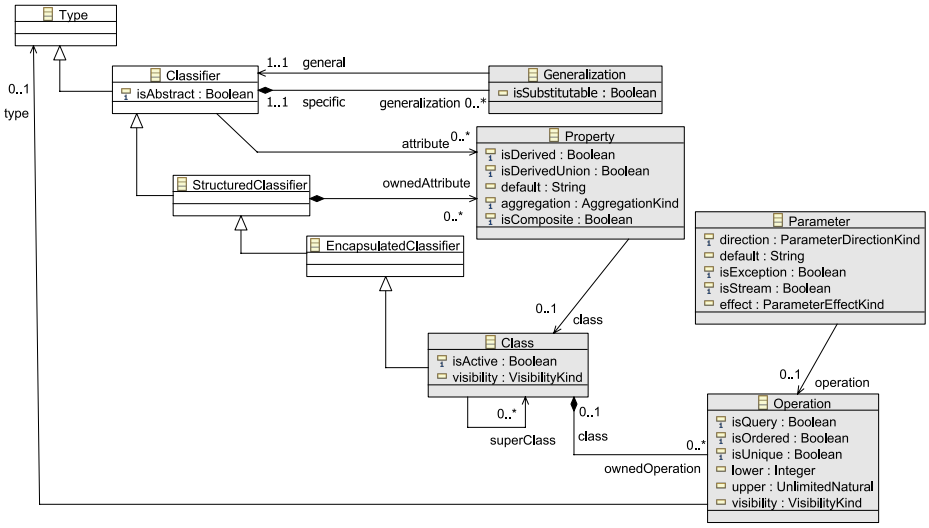

Fig. 4. Subset of the UML Metamodel

metamodel has only one. Another example is the `ClassDefinition` in MOF, which is missing an attribute `visibility` compared to the UML and Java metamodels.

– **Opposites may be missing in relationships**.
  For example, the opposite of the reference related to the notion of inheritance (namely, `superClass` in the MOF and UML metamodels, and `extends` in the Java metamodel) is missing in the three metamodels.
– **The way metamodel classes are linked together may be different** from one metamodel to another.
  For example, the classes `Operation` and `Variable` in the Java metamodel are not directly accessible from `Class` as opposed to the corresponding classes in the MOF and UML metamodels.

Because of these differences among these three metamodels, we are not able to directly reuse a Pull Up Method refactoring accross all three metamodels. Thus, we are forced to write three refactorings, one for each of the three metamodels. In Section 4, we present an approach that allows the specification of one common refactoring for these different metamodels.

3 Kermeta and Model Typing

We introduce here new features of the Kermeta language and the notion of model typing to ease the comprehension of our approach presented in Section 4.

3.1 New Features of Kermeta

In the current version of Kermeta, its action language provides new features for weaving aspects, adding derived properties, and specifying constraints such as invariants and pre-/post-conditions. Indeed, the first new feature of Kermeta is its ability to extend an existing metamodel with new structural elements (classes, operations, and properties) by weaving aspects (similar to inter-type declarations in AspectJ or open-classes [7]). This feature offers more flexibility to developers by enabling them to easily manipulate and reuse existing metamodels while separating concerns. The second new key feature is the possibility to add derived properties. A derived property is a property that is derived or computed through getter and setter accessors for simple types and `add` and `remove` methods for collection types. The derived property thus contains a body, as operations do, and can be accessed in read/write mode. Thanks to this feature, it is possible to figure out the value of a property based on the values of other properties belonging to the same class. The last new feature is the specification of pre- and post-conditions on operations and invariants on classes. These assertions can be directly expressed in Kermeta or imported from OCL (Object Constraint Language) files [28].

3.2 Model Typing

The last version of the Kermeta language integrates the notion of model typing [31], which corresponds to a simple extension to object-oriented typing in a model-oriented context. Model typing can be related to structural typing found in languages such as Scala. Indeed, a model typing is a strategy for typing models as collections of interconnected objects while preserving type conformance, used as a criterion of substitutability.

The notion of model type conformance (or substitutability) has been adapted and extended to model types based on Bruce's notion of type group matching [6]. The matching relation, denoted $<\#$, between two metamodels defines a function of the set of classes they contain according to the following definition:

> Metamodel *M'* matches another metamodel *M* (denoted $M' <\# M$) iff for each class *C* in *M*, there is one and only one corresponding class or subclass *C'* in *M'* such that every property *p* and operation *op* in *M.C* matches in *M'.C'* respectively with a property *p'* and an operation *op'* with parameters of the same type as in *M.C*.

This definition is adapted from [31] and improved here by relaxing two strong constraints. First, the constraint related to the name-dependent conformance on properties and operations was relaxing by enabling their renaming. The second constraint related to the strict structural conformance was relaxing by extending the matching to subclasses.

Let's illustrate model typing with two metamodels *M* and *M'* given in Figures 5 and 6. These two metamodels have model elements that have different names and the metamodel *M'* has additional elements compared to the metamodel *M*.

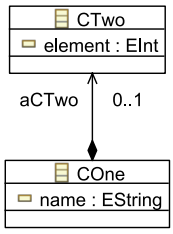

Fig. 5. Metamodel *M*

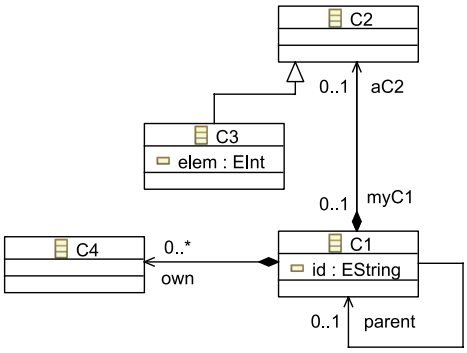

Fig. 6. Metamodel *M'*

C1 <# *COne* because for each property *COne.p* of type *D* (namely, *COne.name* and *COne.aCTwo*), there is a matching property *C1.q* of type *D'* (namely, *C1.id* and *C1.aC2*), such that *D'* <# *D*.

Thus, *C1* <# *COne* requires *D'* <# *D*, which is true because:

- *COne.name* and *C1.id* are both of type *String*.
- *COne.aCTwo* is of type *CTwo* and *C1.aC2* is of type *C2*, so *C1* <# *COne* requires *C2* <# *CTwo* or that a subclass of *C2* matches *CTwo*. Only *C3* <# *CTwo* is true because *CTwo.element* and *C3.elem* are both of type *String*.

Thus, matching between classes may depend on the matching of their related dependent classes. As a consequence, the dependencies involved when evaluating model type matching are heavily cyclical [30]. The interested reader can find in [30] the details of matching rules used for model types.

However, the model typing with the mechanisms of renaming and inheritance is not sufficient for matching metamodels that are structurally different. We show in the next section with our approach how we overcome this limitation of the model typing using aspect weaving.

4 Approach: Specification of Generic Refactorings

In this section, we present our approach for generic model refactoring. The four steps of the approach are illustrated in Figure 7. The first step consists of specifying a generic metamodel `GenericMT`[1], which corresponds to a metamodel that only contains elements required for applying refactorings. The second step consists of specifying refactorings based on the source metamodel `GenericMT` using a model transformation language such as Kermeta. The third step aims to adapt the target metamodels (Java, MOF, and UML) to the metamodel `GenericMT`.

[1] MT refers to Model Type.

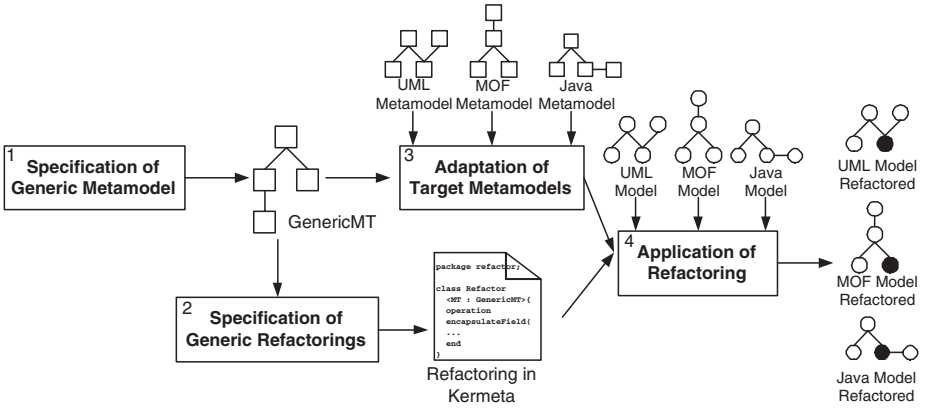

Fig. 7. Approach for the Specification of Generic Refactorings

In the last step, refactorings are directly applied to models of all target metamodels. We detail in the following each of these steps using the Pull Up Method refactoring as a running example.

Step 1: Specification of Generic Metamodel. Our approach consists first of specifying a lightweight metamodel that contains the minimum required classes, methods, and attributes for specifying refactorings. The generic metamodel, called `GenericMT` and given in Figure 8, has been designed to specify refactorings. `GenericMT` contains concepts common to most of object-oriented metamodels such as classes, methods, attributes, and parameters.

Step 2: Specification of Generic Refactorings. In the second step, refactorings are specified based on the generic metamodel `GenericMT`. Listing 1 gives

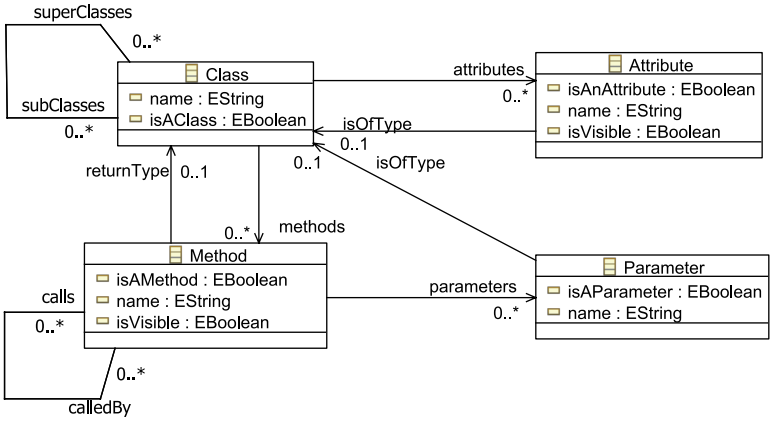

Fig. 8. Generic Metamodel `GenericMT`

a Kermeta code[2] excerpt of the class `Refactor`, which contains the operation `pullUpMethod`. This operation aims to pull up the method `meth` from the source class `source` to the target class `target`. This operation contains a precondition that checks if the sibling subclasses have methods with the same signatures. In the body of the operation, the method `meth` is added to the methods of the target class and removed from the methods of the source class.

```
package refactor;

class Refactor<MT : GenericMT> {

    operation pullUpMethod( source : MT::Class, target : MT::Class,
                            meth   : MT::Method) : Void

    // Preconditions
    pre sameSignatureInOtherSubclasses is do
        target.subClasses.forAll{ sub |
            sub.methods.exists{ op | haveSameSignature(meth, op) } }
    end

    // Operation body
    is do
        target.methods.add(meth)
        source.methods.remove(meth)
    end
}
```

Listing 1. Kermeta Code for the Pull Up Method Refactoring

Step 3: Adaptation of Target Metamodels. The third step aims to adapt the target metamodels to the generic metamodel `GenericMT` using the new Kermeta features for weaving aspects and adding derived properties. The adaptation consists of weaving, in the target metamodels, derived properties that match with those of the generic metamodel. This step of adaptation is necessary because the model typing is too restrictive for allowing a matching between metamodels that are structurally too different. Thus, this adaptation virtually modifies the structure of the target metamodel with additional elements, and uses the model typing to match the metamodels.

The adaptation requires also the weaving of opposites. The opposites are identified in Kermeta by a sharp ♯ and are computed during the loading of the model. The opposites make easier the writing of adapters by adding required navigation links.

Listings 2, 4, and 3 present the adaptations of the derived properties `superClasses` and `subClasses` of `Class` respectively for the Java, MOF, and UML target metamodels given respectively in Figures 2, 3, and 4. Because of lack of space, we provide only the getter accessors of the derived properties; the setter accessors are symmetric.

Adaptation for the Java metamodel. The derived property `superClasses` corresponds to a simple access to the property `extends` that is then wrapped in a Java

[2] The interested reader can refer to the Kermeta syntax in [16].

`Class`. However, for the derived property `subClasses`, the opposite `inv_extends` of the property `extends` was weaved by aspect on the class `Classifier` and used to get the set of subclasses.

```
package java;

require "Java.ecore"

aspect class Classifier {
    reference inv_extends : Classifier[0..*]#extends
    reference extends : Classifier[0..1]#inv_extends
}

aspect class Class {

    property superClasses : Class[0..1]#subClasses
        getter is do
            result:=self.extends
        end

    property subClasses : Class[0..*]#superClasses
        getter is do
            result := OrderedSet<java::Class>.new
            self.inv_extends.each{ subC | result.add(subC) }
        end
}
```

Listing 2. Kermeta Code for Adapting the Java Metamodel

Adaptation for the UML metamodel. In UML, the inheritance links are reified through the class `Generalization`. Thus, the derived property `superClasses` is computed by accessing to the class `Generalization` and the reference property `general`. As in Java and MOF, an opposite `inv_general` is specified to get the set of subclasses.

```
package uml;

require "http://www.eclipse.org/uml2/2.1.2/UML"

aspect class Classifier {
    reference inv_general : Generalization[0..*]#general
}

aspect class Class {

    property superClasses : Class[0..*]#subClasses
        getter is do
            result := OrderedSet<uml::Class>.new
            self.generalization.each{ g | result.add(g.general) }
        end

    property subClasses : Class[0..*]#superClasses
        getter is do
            result := OrderedSet<uml::Class>.new
            self.inv_general.each{ g | result.add(g.specific) }
        end
}
```

Listing 3. Kermeta Code for Adapting the UML Metamodel

```
package kermeta;

require kermeta

aspect class ParameterizedType {
 reference typeDefinition: GenericTypeDefinition[1..1]#
     inv_typeDefinition
}

aspect class GenericTypeDefinition {
 reference inv_typeDefinition: ParameterizedType[1..1]#typeDefinition
}

aspect class Type {
 reference inv_superType: ClassDefinition[0..*]#superType
}

aspect class ClassDefinition {

    reference superType : Type[0..*]#inv_superType

    property superClasses : ClassDefinition[0..*]#subClasses
        getter is do
            result := OrderedSet<ClassDefinition>.new
            self.superType.each{ c |
                var clazz : Class init Class.new
                clazz ?= c
                var clazzDef : ClassDefinition init ClassDefinition.new
                clazzDef ?= clazz.typeDefinition
                result.add(clazzDef) }
        end

    property subClasses : ClassDefinition[0..*]#superClasses
        getter is do
            result := OrderedSet<ClassDefinition>.new
            var clazz : Class
            clazz ?= self.inv_typeDefinition
            clazz.inv_superType.each{ superC | result.add(superC) }
        end
}
```

Listing 4. Kermeta Code for Adapting the MOF Metamodel

Adaptation for the MOF metamodel. Because of the distinction in the MOF between `Type` and `TypeDefinition` to handle the generic types, it is less straightforward to compute the derived properties `superClasses` and `subClasses`. Several opposites are required as shown in Listing 4.

Step 4: Application of Refactoring. The last step of our approach consists of applying the refactoring on the target metamodels as illustrated in Listing 5 for the UML metamodel. We reuse the example of the method `bill` in the LAN application. We can notice that the class `Refactor` takes as argument the UML metamodel, which thanks to the adaptation of Listing 3 is now a subtype of the expected supertype `GenericMT` as specified in Listing 1. The model typing guarantees the type conformance between the UML metamodel and the generic metamodel.

```
package refactor;

require "http://www.eclipse.org/uml2/2.1.2/UML"

class Main {
    operation main() : Void is do

        var rep : EMFRepository init EMFRepository.new

        var model : uml::Model
        model ?= rep.getResource("lan_application.uml").one

        var source : uml::Class init getClass("PrintServer")
        var target : uml::Class init getClass("Node")
        var meth   : uml::Operation init getOperation("bill")

        var refactor : refactor::Refactor<uml::UmlMM>
                       init refactor::Refactor<uml::UmlMM>.new

        refactor.pullUpMethod(source, target, meth)
    end
}
```

Listing 5. Kermeta Code for Applying the Pull Up Method Refactoring on the UML metamodel

5 Experiments and Discussion

We specified three well known refactorings (Encapsulate Field, Move Method, and Pull Up Method [10]) on models of the LAN application [15] conforming to three different metamodels (Java, MOF, and UML). We were able to successfully apply our approach on these metamodels although they were structurally different. We experimented also a fourth metamodel, which a subset is given in Figure 9. In this metamodel, the two classes (corresponding to `Class` and `Parameter` in the generic metamodel) are unified in a same class (`Type`). This case introduced an ambiguous matching with the generic metamodel since these classes are distinct in the latter. This special case illustrates a limitation of our approach that needs to be overcome and will be investigated in future work. Thus, the only prerequisite of our approach is that each element in the generic metamodel should correspond to a distinct element in the target metamodel. The approach is thus not very restrictive since the mechanism of adaptation enables to raise the inherent limitations of metamodels.

Our approach theoretically relies on the model typing and is feasible in practice thanks to the mechanism of adaptation. Writing adaptations can be more or less difficult depending on the developers' knowledge of the target metamodels. However, once the adaptation is done, the developers can reuse all model refactorings written for the generic metamodel. Conversely, if a developer specifies a new refactoring on the generic metamodel, it can readily be applied on all target metamodels if adaptations are provided.

Although we use a specific kind of model transformations, namely refactorings, for demonstrating the feasibility of our approach, this one can be applied to any other endogenous model transformation. In addition, our approach also fits well in the context of metamodel evolution. Indeed, all model transformations written

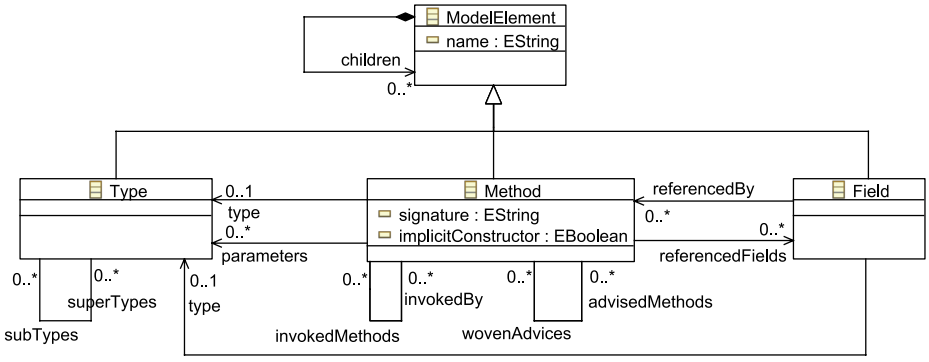

Fig. 9. Subset of the Fourth Metamodel

for an old version of a given metamodel (for example, UML 1.2) can be reused for a new version (for example, UML 2.0) once the adaptation is done. Moreover, the models do not need to be migrated from the old version to the new one. Finally, our approach (with the model typing, the mechanism of adaptation, and the generic metamodel) can be seen as a framework for specifying arbitrary model transformations for arbitrary metamodels.

6 Related Work

Genericity and reuse in MDE have not been sufficiently investigated as in object-oriented (OO) programming. However, we observe some efforts in the MDE community that are directly inherited from type-safe code reuse in OO programming and, in particular, from generic programming.

Generic programming is about making programs more adaptable by making them more general [11]. This style of programming allows writing programs that differ in their parameters, which may be either other programs, types and type constructors, class hierarchies, or even programming paradigms [11]. Aspects [17] and open-classes [7] are powerful generic programming techniques for adapting programs by augmenting their behavior in existing classes [12,18]. Similarly, in our approach, we use aspects to align target metamodels with the generic metamodel. Other languages that provide support for generic programming are Haskell and Scala [26]. The use of Haskell has been investigated [21] to specify refactorings based on high level graph algorithms that could be generic accross a variety of languages (XML, Pascal, Java), but its applicability does not seem to go beyond a proof of concept. Scala's implicit conversions [9] simulate the open-class mechanism in order to extend the behavior of existing libraries without actually changing them. Although Scala is not a *model-oriented* language, developers can build type-safe reusable model transformations on top of EMF thanks to its good integration with Java. However, it would require to write a significant amount of code and manage relationships among generic types.

In the MDE community, Blanc *et al.* proposed an architecture, called Model Bus, that allows the interoperability of a wide range of modeling services [4]. The

term '*modeling service*' defines an operation having models as inputs and outputs such as model edition, model transformation, and code generation. Their architecture is based on a metamodel that ensures type compatibility checking by describing services as software components having precise input and output definitions. However, the type compatibility defined in this metamodel relies on a simple notion of model types as sets of metaclasses, but without any notion of model type substitutability. Other work [1,24] study the problem of generic model transformations using a mechanism of parameterization. However, these transformations do not apply to different metamodels but to a set of related models.

Modularity in graph transformation systems was also explored [13]. In this area, an interesting work was done by Engels *et al.* who presented a framework for classifying and defining relations between typed graph transformation systems [8]. This framework integrates a novel notion of substitution morphism that allows to define the semantic relation between the required and provided interfaces of modules in a flexible way.

From another perspective, our approach also relates to the Aspect Oriented Modeling (AOM) field [19], or more precisely to AO metamodeling. From the AOM perspective, our notion of model type can indeed be interpreted as a *pointcut* defining a (model) pattern to be matched in a specific metamodel (e.g. UML or Java). The definition of our generic refactorings then would play the role of *advices* to be woven into these metamodels through some kind of adaptation as available in SmartAdapters [20]. Thus from this perspective our paper could have been title "Weaving refactoring aspects into metamodels".

7 Conclusion

In this paper, we have presented an approach for generic model refactorings, that is refactorings that can be reused on structurally different metamodels. This approach relies on the model typing and a mechanism of adaptation based mainly on the weaving of aspects. We illustrated our approach on the Pull Up Method refactoring and validated it on three different refactorings (Encapsulate Field, Move Method, and Pull Up Method) for three different metamodels (Java, MOF, and UML) in a concrete application. We demonstrated that our approach ensures a flexible reuse of model transformations, in particular refactorings. This approach seems to be generalisable to other endogenous model transformations such as the computation of metrics, detection of patterns and inconsistencies. As future work, we plan to increase the repository of refactorings on other metamodels and experiment with other model transformations.

References

1. Amelunxen, C., Legros, E., Schurr, A.: Generic and reflective graph transformations for the checking and enforcement of modeling guidelines. In: Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2008), Washington, DC, USA, pp. 211–218. IEEE Computer Society, Los Alamitos (2008)

2. Basili, V.R., Briand, L.C., Melo, W.L.: How reuse influences productivity in object-oriented systems. Communications of ACM 39(10), 104–116 (1996)
3. Biggerstaff, T.J., Perlis, A.J.: Software Reusability Volume I: Concepts and Models, vol. I. ACM Press, Addison-Wesley, Reading (1989)
4. Blanc, X., Gervais, M.-P., Sriplakich, P.: Model bus: Towards the interoperability of modelling tools. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 17–32. Springer, Heidelberg (2005)
5. Blanc, X., Ramalho, F., Robin, J.: Metamodel reuse with mof., pp. 661–675 (2005)
6. Bruce, K.B., Vanderwaart, J.: Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. Electronic Notes in Theoretical Computer Science 20, 50–75 (1999)
7. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.D.: Multijava: Modular open classes and symmetric multiple dispatch for java. In: Proceedings of the 15^{th} International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 130–145 (2000)
8. Engels, G., Heckel, R., Cherchago, A.: Flexible interconnection of graph transformation modules. In: Kreowski, H.-J., Montanari, U., Orejas, F., Rozenberg, G., Taentzer, G. (eds.) Formal Methods in Software and Systems Modeling. LNCS, vol. 3393, pp. 38–63. Springer, Heidelberg (2005)
9. Odersky, M., et al.: An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)
10. Fowler, M.: Refactoring – Improving the Design of Existing Code, 1st edn. Addison-Wesley, Reading (1999)
11. Gibbons, J., Jeuring, J. (eds.): Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, Dagstuhl, Germany, July 11-12. IFIP Conference Proceedings, vol. 243. Kluwer Academic Publishers, Dordrecht (2003), `http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/wcgp-preface.pdf`
12. Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectj. SIGPLAN Not. 37(11), 161–173 (2002)
13. Heckel, R., Engels, G., Ehrig, H., Taentzer, G.: Classification and comparison of module concepts for graph transformation systems. In: Handbook of graph grammars and computing by graph transformation. Applications, languages, and tools, vol. 2, pp. 669–689. World Scientific, Singapore (1999)
14. Hoffman, B., Pérez, J., Mens, T.: A case study for program refactoring. In: GraBaTs (September 2008)
15. Janssens, D., Demeyer, S., Mens, T.: Case study: Simulation of a lan. Electronic Notes in Theoretical Computer Science 72(4) (2003)
16. Kermeta, `http://www.kermeta.org/`
17. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
18. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Proceedings of the 27^{th} international conference on Software engineering (ICSE 2005), pp. 49–58. ACM, New York (2005)
19. Kienzle, J., Abed, W.A., Jacques, K.: Aspect-oriented multi-view modeling. In: AOSD 2009: Proceedings of the 8th ACM international conference on Aspect-oriented software development, pp. 87–98. ACM, New York (2009)

20. Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., Jézéquel, J.-M.: Introducing variability into aspect-oriented modeling approaches. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 498–513. Springer, Heidelberg (2007)
21. Lämmel, R.: Towards Generic Refactoring. In: Proceedings of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE 2002, Pittsburgh, USA, October 5, 14 pages. ACM Press, New York (2002)
22. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
23. Mili, H., Mili, F., Mili, A.: Reusing software: Issues and research directions. IEEE Transactions of Software Engineering 21(6), 528–562 (1995)
24. Münch, M.: Generic Modelling with Graph Rewriting Systems. PhD thesis, RWTH Aachen, Berichte aus der Informatik (2003)
25. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
26. Oliveira, B.C.D.S., Gibbons, J.: Scala for generic programmers. In: Hinze, R., Syme, D. (eds.) WGP 2008: Proceedings of the ACM SIGPLAN workshop on Generic programming, pp. 25–36. ACM, New York (2008)
27. OMG. Mof 2.0 core specification. Technical Report formal/06-01-01, OMG, April 2006. OMG Available Specification
28. OMG. The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07 (2007)
29. OMG. The uml 2.1.2 infrastructure specification. Technical Report formal/2007-11-04, OMG, April 2007. OMG Available Specification
30. Steel, J.: Typage de modèles. PhD thesis, Université de Rennes (April 1, 2007)
31. Steel, J., Jézéquel, J.-M.: On model typing. Journal of Software and Systems Modeling (SoSyM) 6(4), 401–414 (2007)