Software and System Modeling manuscript No.
(will be inserted by the editor)

Towards Reusable Model Transformations ⋆

Sagar Sen, Naouel Moha, Vincent Mahé, Benoit Baudry, Olivier Barais, Jean-Marc Jézéquel

INRIA Rennes - Bretagne Atlantique / IRISA, Université Rennes 1

Triskell Team, Campus de Beaulieu, 35042 Rennes Cedex, France

{e-mail: ssen, moha, vmahe, bbaudry, barais, jezequel}e-mail: @irisa.fr

Received: date / Revised version: date

Abstract Very often model transformations written for an input metamodel apply to other metamodels that share similar concepts. For example, a transformation written to refactor Java models can be applicable to refactoring UML class diagrams as both Java and UML share concepts such as of classes, methods, attributes, and inheritance. Another example is where a company needs to economically upgrade its legacy transformations from conforming to an old metamodel to a similar new industry standard metamodel such as the latest UML. The old metamodel may either be from an in-house DSML or an old version an industry standard such as UML. Deriving motivation from these examples we present an approach to make model transformations *reusable* such that they function correctly across several similar metamodels. Our approach relies on these principal steps: (1) We generate an effective input metamodel via transformation analysis and metamodel pruning of the input metamodel. The effective metamodel represents the true input domain of the transformation (2) We adapt a target input metamodel by weaving it with aspects to make it a *subtype* of the effective input metamodel. The subtype property ensures that the transformation can process models conforming to the target input metamodel. We validate our approach by adapting well-known refactoring transformations (Encapsulate Field, Move Method, and Pull Up Method) written for an in-house DSML to three different industry standard metamodels (Java, MOF, and UML).

Key words Adaptation, Aspect Weaving, Genericity, Model Typing, Refactoring, Metamodel Pruning

1 Introduction

Model transformations are software artifacts that underpin complex software system development in Model-driven Engineering (MDE). Making model transformations *reusable* is the subject of this paper.

Software reuse in general has been largely investigated in the last two decades by the software engineering community [1,2]. Basili *et al.* [3] demonstrate the benefits of software reuse on the productivity and quality in object-oriented systems. However, reuse is a new entrant in the MDE scenario [4]. One of the primary difficulties in making a model transformation reusable across different input domains is the difference in structural aspects between commutable input metamodels. Consider an example where model transformation reuse becomes obvious and yet is infeasible due to structural differences in commutable input metamodels. A model transformation to *refactor models of class diagrams* is possible in several modelling languages supporting the concepts of classes, methods, attributes, and inheritance. For instance, the metamodels for the languages Java, MOF, and UML all contain concepts needed to specify class diagram models. If we emphasize the necessity for reuse then the refactoring transformation must be intuitively adaptable to all three metamodels : Java, MOF, and UML as they manipulate similar models. Hence, we ask: How do we reuse one implementation of a model transformation for other possibly similar modelling languages? This is the question that intrigues us and for which we provide a solution.

Our aim is to enable flexible reuse of model transformations across various metamodels to enhance productivity and quality in MDE. In this paper, we present an approach to make legacy model transformations reusable for different target input metamodels. We do not touch the body of the legacy transformation itself but transform a target input metamodel such that it becomes a *subtype* of the effective subset of the input metamodel. We call the effective subset an *effective input metamodel* which represents the true input domain of the legacy model transformation. The subtype property permits the legacy model transformation to process pertinent models conforming to the target input metamodel. Concisely, our approach follows these steps: (1) We automatically obtain an effective input metamodel via *metamodel pruning* [5]. This step drastically reduces the adaptation effort in the next step when dealing with large metamodels such as the UML where model transformations often use only a small subset of the entire metamodel (2) We adapt a target input metamodel by weaving it with structural aspects from the effective input metamodel. We also weave accessor function for these structural aspects that seek information from related concepts in the target metamodel (3) We use *model typing* [6] to verify the type conformance between the woven target input metamodel and the effective input metamodel. The woven target input metamodel must be a *subtype* of the effective input metamodel (4) Replacing the original input metamodel with the woven target input metamodel at *run-time* allows the legacy model transformation to process relevant input models conforming to the target input metamodel. The scientific contribution in our approach is based on a combination of two recent ideas namely metamodel pruning [5] and manual specification of generic model refactorings [7]. In [7], the authors manually specify a generic model transformation for handmade generic metamodel that is adapted to various target input metamodels. In our work we automatically synthesize an effective input metamodel via metamodel pruning which is in contrast to manually specifying a generic metamodel as in [7]. Further, the effective input metamodel is derived from an arbitrary input metamodel of a legacy model transformation and not from a domain-specific generic metamodel (for refactoring) as in [7]. The adaptation of target input metamodels to the effective input metamodel via aspect-weaving remains similar to the approach in [7].

We demonstrate our approach on well known model transformations, namely refactorings [8]. 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 [8]. 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 [8]. We validate our approach by performing some experiments where three well known legacy refactorings (Encapsulate Field, Move Method, and Pull Up Method) are adapted to three different industrial metamodels (Java, MOF, and UML). The legacy refactorings are written in Kermeta which is a modelling language to specify structure and behavior of models [9].

This article is organized as follows. In Section 2, we describe a motivating example with arising problems. In Section 3, we introduce foundations necessary to describe our approach. The foundations include a description of the executable metamodeling language, Kermeta, highlights some of its new features including the notion of model typing, and presents meta-model pruning to obtain an effective input metamodel. The Section 4 gives a general step-by-step overview of our approach. The Section 5 describes the experiments that we performed for adapting legacy three refactoring transformations (Encapsulate Field, Move Method, and Pull Up Method) initially described for an in-house DSML to three different industry standard metamodels (Java, MOF, and UML). Section 6 surveys related work. Section 7 concludes and presents future work.

2 Motivating Example

We present an example of a model transformation that performs refactoring on an in-house DSML for modelling software structure and behaviour. Our ultimate objective is to make this model transformation reusable and applicable across different industry standard metamodels. Specifically, we describe the Pull Up Method refactoring transformation which we intend to use for models from 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 [8]. 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 [10] of a Local Area Network (LAN) application [11] and adapted in Figure 1 shows that the method `bill` located in the classes `PrintServer` and `Workstation` is pulled up to their superclass `Node`.

The Pull Up Method refactoring is written for an in-house DSML for the INRIA team TRISKELL from Rennes, France that contains the notions of classes, properties, inheritance, operations and several other concepts related to contracts and verification that are not pertinent to refactoring. The in-house DSML does not conform to an industry standard metamodel such as UML.

2.2 Three Different Metamodels

Our goal is to make the refactoring reusable across three different target input metamodels (Java, MOF, and UML), which support the definition of object-oriented structures (classes, methods, attributes, and inheritance). The Java metamodel described in [12] 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 [9], which is an extension of MOF [13] 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 [14]. 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.

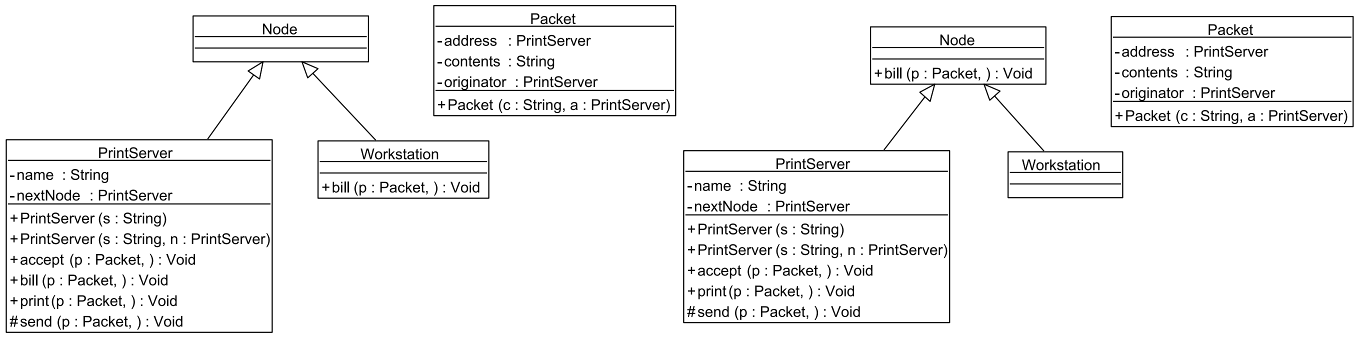
Node
Packet
-address : PrintServer
-contents : String
-originator : PrintServer
+Packet (c : String, a : PrintServer)
PrintServer
-name : String
-nextNode : PrintServer
+PrintServer (s : String)
+PrintServer (s : String, n : PrintServer)
+accept (p : Packet,) : Void
+bill (p : Packet,) : Void
+print (p : Packet,) : Void
#send (p : Packet,) : Void
Workstation
+bill (p : Packet,) : Void
Node
+bill (p : Packet,) : Void
Packet
-address : PrintServer
-contents : String
-originator : PrintServer
+Packet (c : String, a : PrintServer)
PrintServer
-name : String
-nextNode : PrintServer
+PrintServer (s : String)
+PrintServer (s : String, n : PrintServer)
+accept (p : Packet,) : Void
+print (p : Packet,) : Void
#send (p : Packet,) : Void
Workstation

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

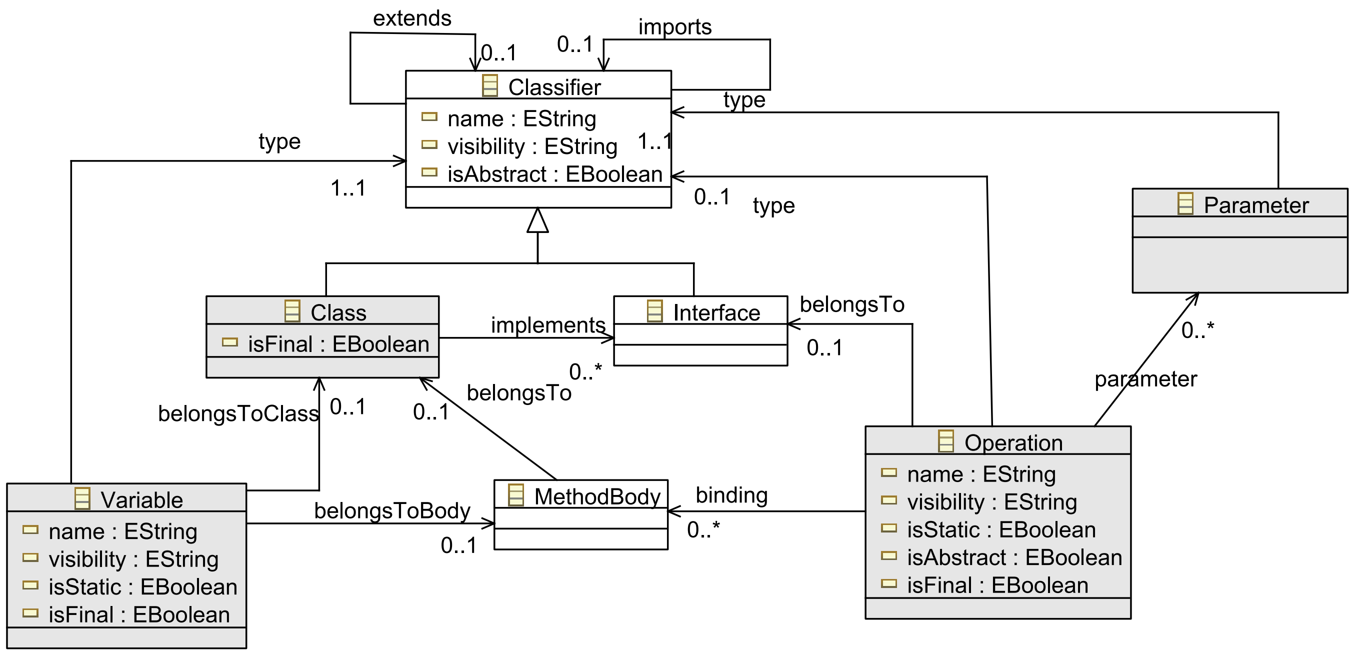

Figure 2 Subset of the Java Metamodel.

2.3 Problems

We encounter several problems if we intend to specify a 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`.
- **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 metamodel has only one. Another

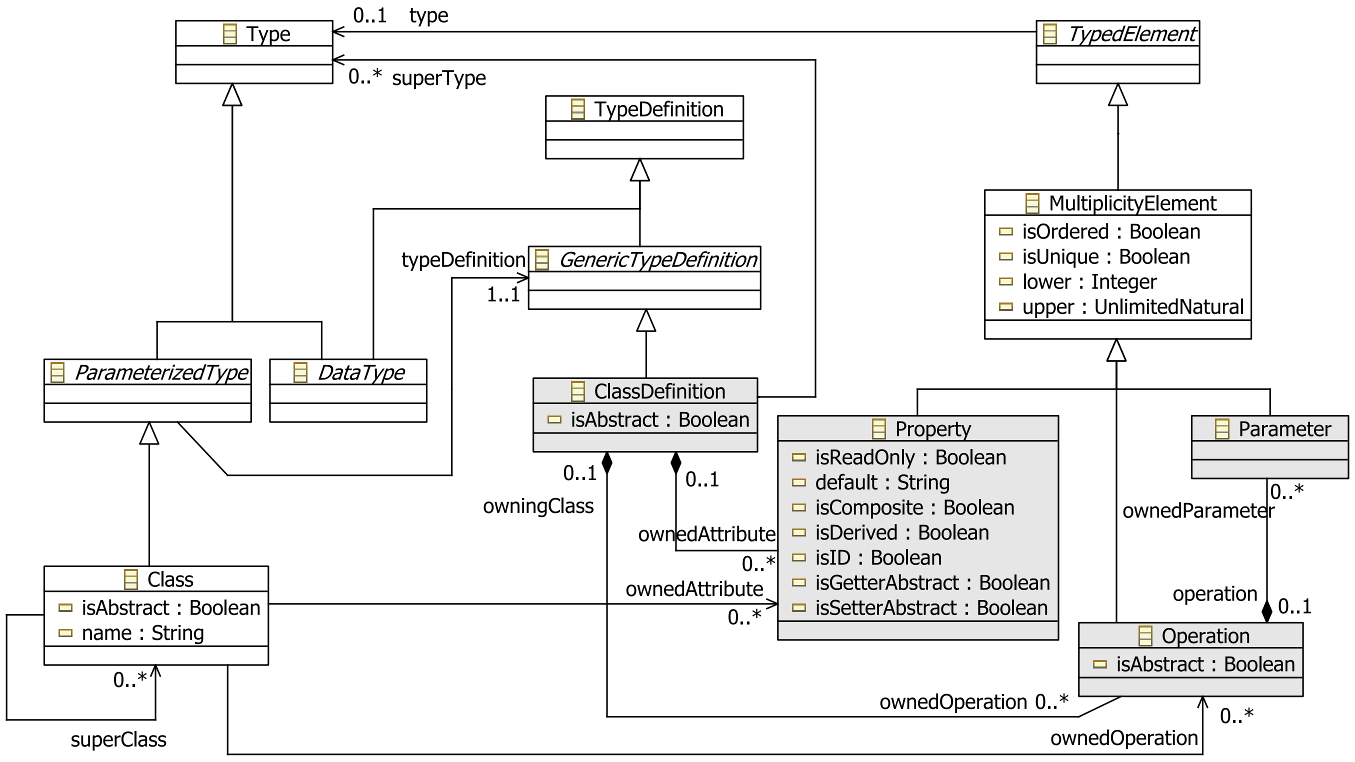

Figure 3 Subset of the MOF Metamodel.

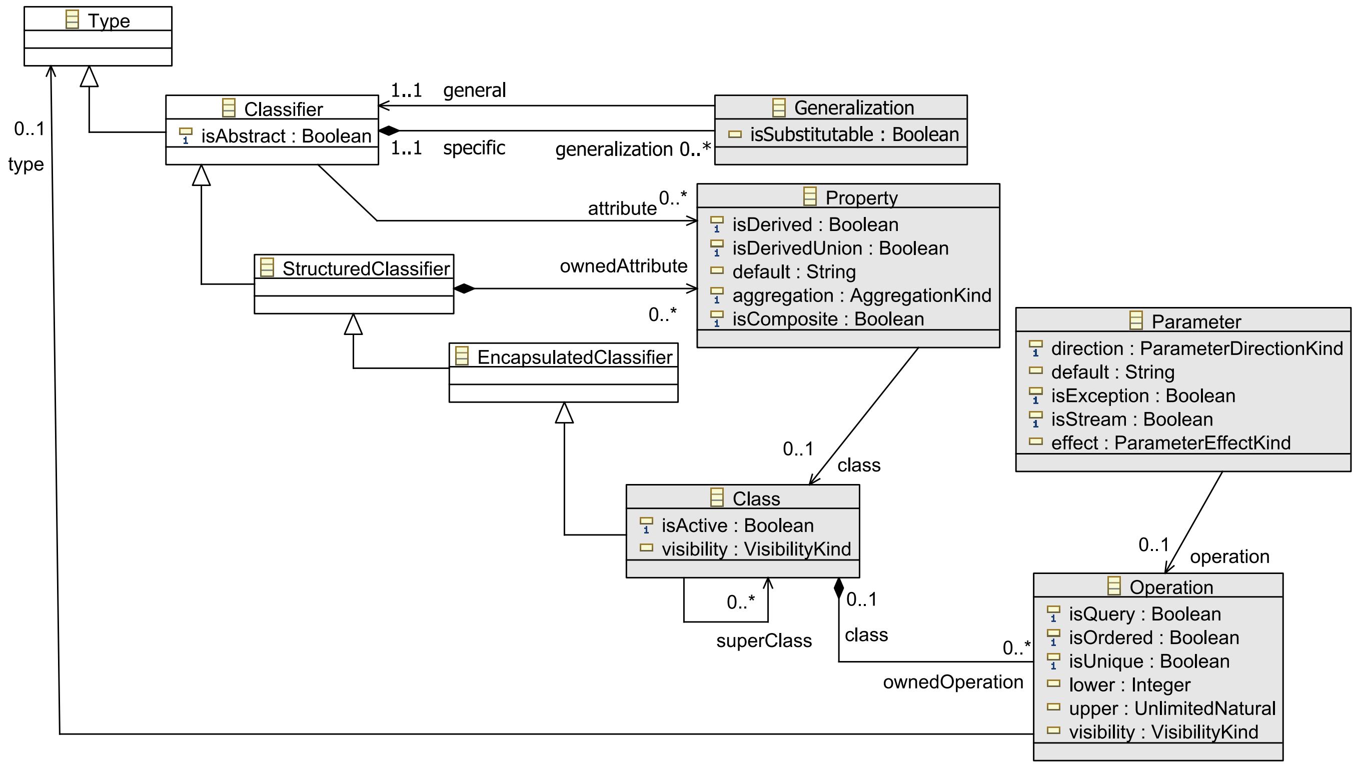

Figure 4 Subset of the UML Metamodel.

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.

These differences among these three metamodels make it impossible to directly reuse a Pull Up Method refactoring across all three metamodels. Hence, we are forced to write three different implementations of the same refactoring transformation for each of the three metamodels. We address this problem with out approach In Section 4. In the approach we make a single transformation reusable across different metamodels without rewriting the transformation. We only adapt different target input metamodels such that they become a subtype of the input metamodel of the transformation.

3 Foundations

This section presents the foundations required to explain the approach presented in Section 4. We describe the model transformation language Kermeta in Section 3.1. We present new features of Kermeta that allow weaving aspects into target input metamodels in Section 3.2. We describe Kermeta's implementation of model typing in Section 3.3 which helps us perform all type conformance operations in our approach. Finally, in Section 3.4 we present the metamodel pruning algorithm to obtain the effective input metamodel to be used in the approach.

3.1 Kermeta

Kermeta is a language for specifying metamodels, models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [13]. The object-oriented meta-language MOF supports the definition of metamodels in terms of object-oriented structures (packages, classes, properties, and operations). It also provides model-specific constructions such as containments and associations between classes. Kermeta extends the MOF with an *imperative action language* for specifying constraints and operational semantics for metamodels [9]. Kermeta is built on top of EMF within the ECLIPSE development environment. The action language of Kermeta provides mechanisms for dynamic binding, reflection, and exception handling. It also includes classical control structures such as blocks, conditionals, and loops.

We note that Kermeta is used to specify the refactorings used in our examples in Section 5.

3.2 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 [15]). 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 [16].

3.3 Model Typing

The last version of the Kermeta language integrates the notion of model typing [6], 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 [17]. 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 [6] 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*.

> *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 [18]. The interested reader can find in [18] 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 overcome this limitation of the model typing by weaving required aspects as described in our approach in Section 4.

3.4 Metamodel Pruning

In our approach of Section 4 we obtain an effective input metamodel from possibly large input metamodels such as that of UML via metamodel pruning [5]. Metamodel pruning conserves a set of required classes and properties and their obligatory dependencies in a metamodel and prunes everything else. The result is an effective metamodel that is a *supertype* of the initial metamodel which can be verified using model typing [6]. In this section we concisely describe the metamodel pruning algorithm.

Given a possibly large metamodel such as UML that may represent the input domain of a model transformation we ask the question : Does the model transformation process models containing objects of all possible types in the input metamodel? In several cases the answer to this question may be no. For instance, a transformation that refactors UML models only processes objects with types that come from concepts in the UML class diagrams subset but not UML Activity, UML Statechart, or UML Use case. How do we obtain this effective subset? This is the problem that metamodel pruning solves.

The principle behind pruning is to preserve a set of required types T_{req} and and required properties P_{req} and

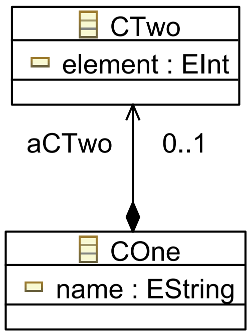

Figure 5 Metamodel *M*.

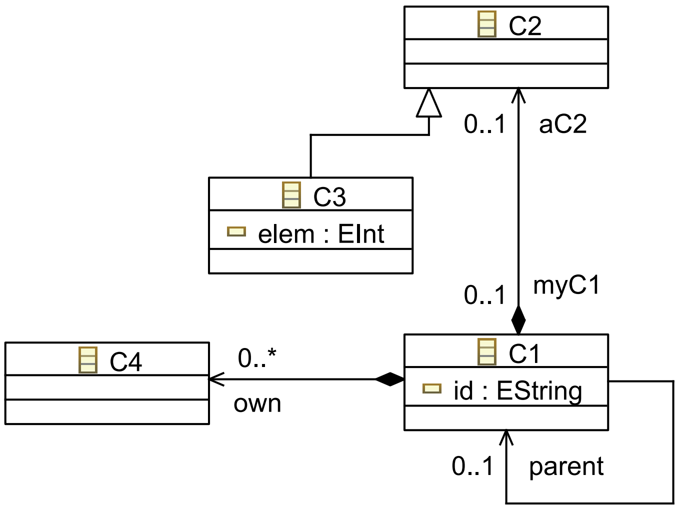

Figure 6 Metamodel *M'*.

prune away the rest in a metamodel. The authors of [5] present a set of rules that help determine a set of required types T_{req} and required properties P_{req} given a metamodel MM and an initial set of required types and properties. The initial set may come from various sources such as manual specification or a static analysis of a model transformation to reveal used types. A rule in the set for example adds all super classes of a required class into T_{req}. Similarly, if a class is in T_{req} or is a required class then for each of its properties p, add p into P_{req} if the lower bound for its multiplicity is > 0. Apart from rules the algorithm contains options which allow better control of the algorithm. For example, if a class is in T_{req} then we add all its sub-classes into T_{req}. This optional rule is not obligatory but may be applicable under certain circumstances giving the user some freedom. The rules are executed where the conditions match until no rule can be executed any longer. The algorithm terminates for a finite metamodel because the rules do not remove anything from the sets T_{req} and P_{req}.

Once we compute the sets T_{req} and P_{req} the algorithm simply removes the remaining types and properties to output the effective metamodel MM_e. The effective metamodel MM_e generated using the algorithm in [5] has some very interesting characteristics. Using model typing (discussed in Section 3.3) we verify that MM_e is a *supertype* of the metamodel MM. This implies that all operations written for MM_e are valid for the large metamodel MM.

4 Approach

We present an approach to make a legacy model transformation `MT` reusable. We outline the approach in Figure 7 and describe the steps in the approach below:

Step 1: Static Analysis of a Transformation

As shown in Figure 7 we first perform *static analysis* on the legacy model transformation `MT`. The static

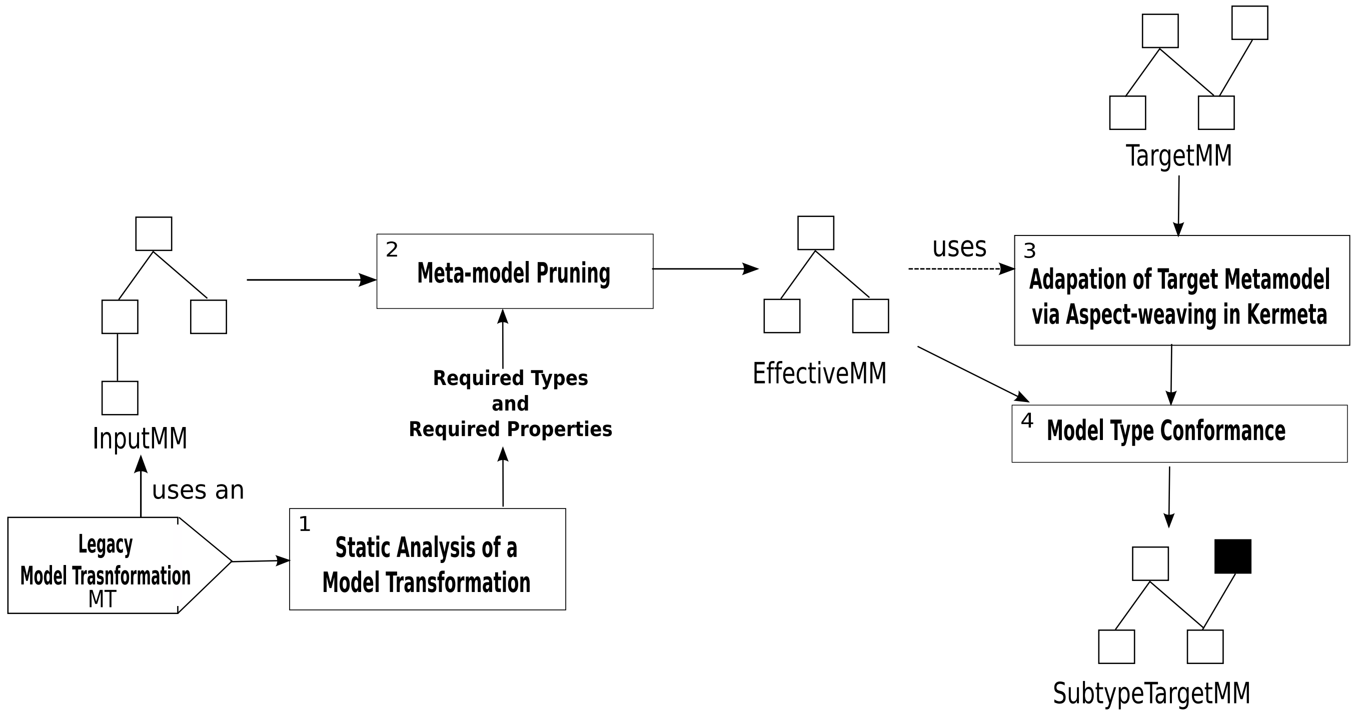
TargetMM
2 Meta-model Pruning
uses
3 Adapation of Target Metamodel via Aspect-weaving in Kermeta
Required Types and Required Properties
EffectiveMM
4 Model Type Conformance
InputMM
uses an
Legacy Model Trasnformation MT
1 Static Analysis of a Model Transformation
SubtypeTargetMM

Figure 7 Approach for Transforming an Input Metamodel to Subtype Target Input Metamodel

analysis involves visiting each rule, each constraint, and each statement in the model transformation to obtain an initial set of required types T_{req} and a set of required properties P_{req} manipulated in the input metamodel `InputMM`. The goal behind performing static analysis is to find the *subset of concepts* in the input metamodel *actually used in the transformation.* We do not go into the details of the static analysis process as it is just classical traversal of the *abstract syntax tree* of an entire program or a rule in order to check the type of each term. If the type is present in `InputMM` we add it to T_{req}. Similarly, we add all properties manipulated and existing in `InputMM` into P_{req}.

Step 2: Meta-model Pruning

Using the set of required types T_{req} and properties P_{req} we perform metamodel pruning on `InputMM` to obtain an effective input metamodel `EffectiveMM` that is a *supertype* of `InputMM`. We recall the metamodel pruning algorithm described in Section 3.4. The algorithm generates the minimal effective input metamodel `EffectiveMM` that contains the required types and properties and their obligatory dependencies. The advantages of automatically obtaining the `EffectiveMM` are the following:

– The `EffectiveMM` represents the true input domain of the legacy model transformation `MT`
– The `EffectiveMM` containing only relevant concepts from the `InputMM` drastically reduces the number of aspect-weaving and type matching operations to be performed in Step 4. There is often a combinatorial explosion in the number of type comparisons given that each concept in the target metamodel must be compared with the input metamodel.

The metamodel pruning process plays a key role when the input domain of a transformation corresponds to an Object Management Group standard metamodel such as the UML where the number of classes is about 243 and properties about 587. Writing adaptations for each of these classes as we shall see in Step 3 become very tedious and is not required when only a subset of the input metamodel is in use.

Step 3: Aspect-weaving of Target Metamodel

We recall that in Section 3.2 we present some new features of Kermeta. One of the new features of Kermeta is to be able to weave aspects into metamodels. In the third step we *manually* identify and weave aspects from `EffectiveMM` into the `TargetMM`. We also weave *getter* and *setter* accessor functions into `TargetMM`. These accessors seek information in related concepts of the `TargetMM` and assigns their values to the initially woven properties and types from `EffectiveMM` . We verify the subtype property as described in Step 4. Examples of woven aspects are given in Section 5.

Step 4: Model Type Conformance

We perform model type conformance between the effective input metamodel `EffectiveMM` and the target input metamodel `TargetMM` with woven properties. The model type matching process is described in Section 3.3. All the types in the woven `TargetMM` are matched against each type in `EffectiveMM`. If all types match then `TargetMM` with aspects is the subtype target input metamodel: `SubTypeTargetMM`. Replacing the input metamodel of the legacy model transformation `MT` with `SubTypeTargetMM` will allow all pertinent models conforming to the target input metamodel to be processed by `MT` as shown in Figure 8. When we say pertinent models we mean all models containing objects with types conforming to the used types in the input metamodel of `MT`.

5 Experiments and Discussion

We perform an experiment by applying our approach to legacy model refactoring transformation written for an in-house DSML to three industry standard metamodels Java, UML, and MOF. A step-by-step application of our approach is described in Section 5.1. We discuss the experiment in Section 5.2.

5.1 Application

In Step 1, we perform static analysis of refactoring model transformations applied on an in-house DSML. The result of the static analysis is a set of required types and required properties. The analysis reveals that required classes in the transformation are : `Class`, `Attribute`, `Method`, and `Parameter`. The DSML contains several other classes for Statechart modelling, verification, and activities. These classes and their properties are not used by the refactoring transformation and hence the static analysis does not reveal them. Due to space limitations we do not show the entire DSML in the paper.

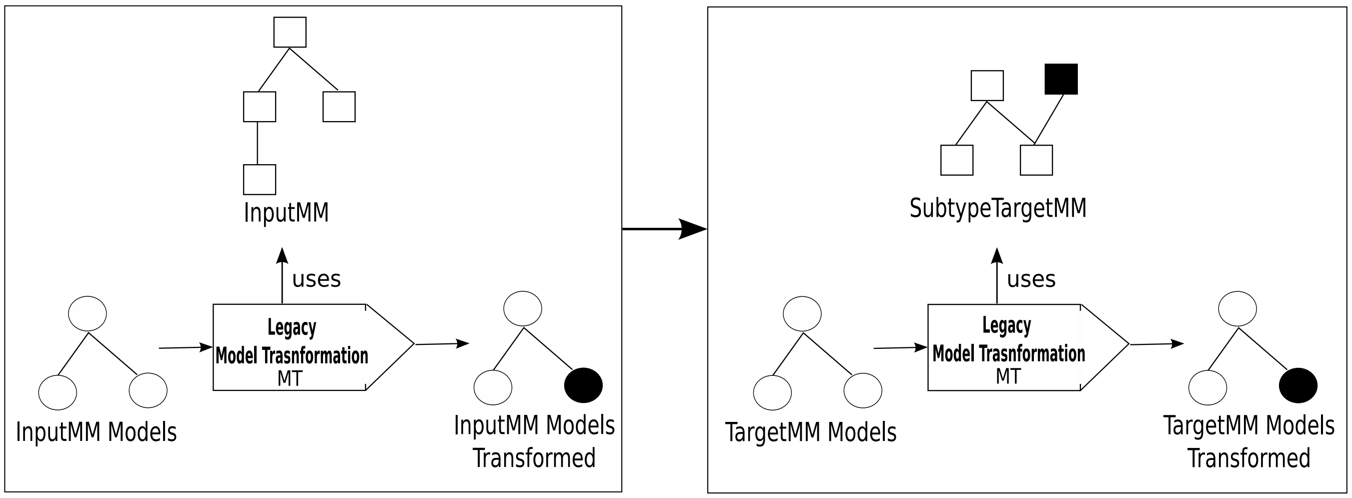

Figure 8 The Legacy Transformation used as a Generic Transformation for TargetMM

We present an example of a refactoring transformation in Listing 1 which is an excerpt of the class `Refactor`. The class `Refactor` contains the operation `pullUpMethod`. The refactoring is implemented in Kermeta [1] 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 : EffectiveMM>
{
    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.

[1] The interested reader can refer to the Kermeta syntax in [19].

In Step 2, we perform metamodel pruning of the input metamodel `InputMM` for the refactoring transformation. We show the resulting effective input metamodel `EffectiveMM` in Figure 9. As claimed earlier the effective metamodel only contains the required types, required properties and their obligatory dependencies. The only inputs to the metamodel pruning algorithm were the classes `Class`, `Attribute`, `Method`, and `Parameter`. The rest of the obligatory structure for the `EffectiveMM`

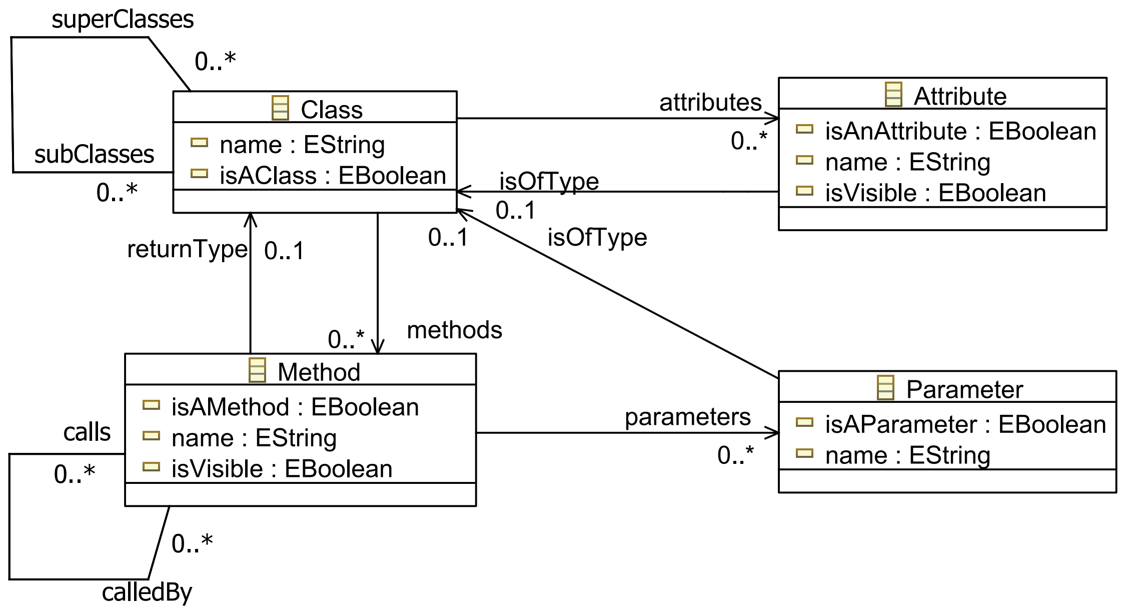

Figure 9 Effective Metamodel `EffectiveMM`.

metamodel is automatically conserved by the metamodel pruning algorithm. All other irrelevant classes for statecharts, verification, and activities are automatically removed.

In Step 3, we adapt the target input metamodels to the effective input metamodel `EffectiveMM` using the new Kermeta features for weaving aspects and adding derived properties. The adaptation consists of weaving the target input metamodels with derived properties that match those in the effective input metamodel. 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 ease the writing of adapters by adding required navigation links. The adaptation also weaves getter and setter accessors that seek information in the `TargetMM` to assign values to the derived properties woven in from `EffectiveMM`. This step of adaptation is necessary because model typing is too restrictive for allowing a matching between metamodels that are structurally too different. The adaptation virtually modifies the structure of the target input metamodel with additional elements and in the following step we use model typing to match the metamodels. The resulting subtype target input metamodel is $SubTypeTargetMM$.

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 input 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 `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: EffectiveTypeDefinition
     [1..1]#inv_typeDefinition
}
aspect class EffectiveTypeDefinition {
 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. Due to 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.

In Step 4, of our approach consists of applying the refactoring on the target input 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 `EffectiveMM` as specified in Listing 1. The model typing guarantees the type conformance between the UML metamodel and the effective input 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.

5.2 Discussion

We reuse three well known refactorings (Encapsulate Field, Move Method, and Pull Up Method [8]) on models of the LAN application [11] conforming to three different metamodels (Java, MOF, and UML). We were able to successfully apply our approach on these metamodels

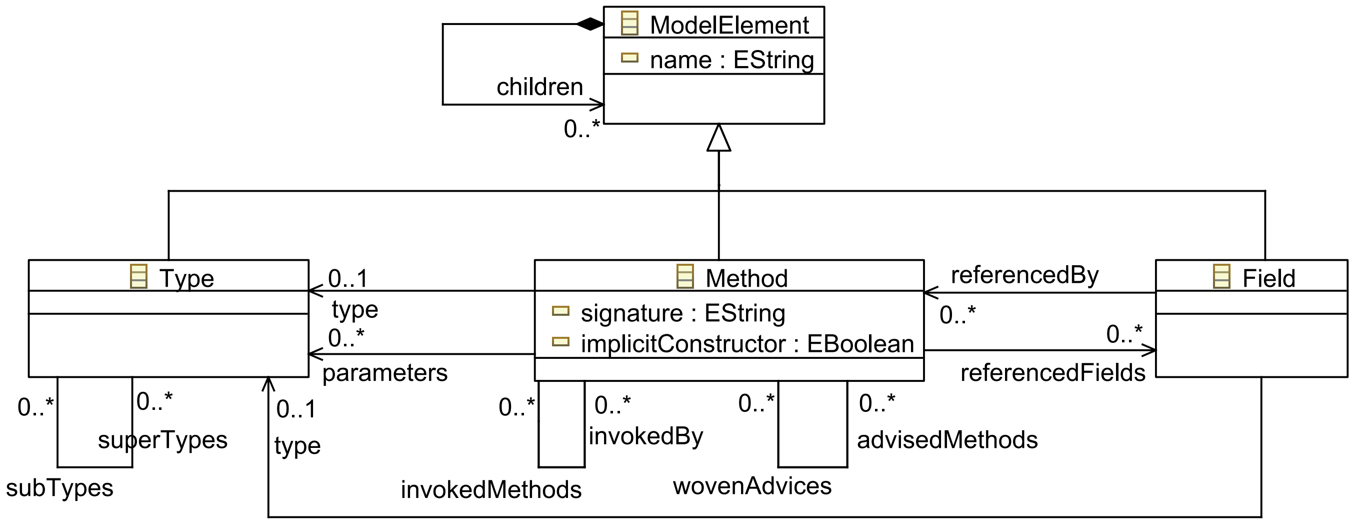

Figure 10 Subset of the Fourth Metamodel.

although they were structurally different. We experimented also a fourth metamodel as shown in Figure 10. In this metamodel, the two classes (corresponding to `Class` and `Parameter` in the effective input metamodel) are unified in a same class (`Type`). This case introduced an ambiguous matching with the effective input 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 effective input metamodel should correspond to a distinct element in the target input 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 input metamodels. However, once the adaptation is done, the developers can reuse all model refactorings written for the original input metamodel. Conversely, if a developer specifies a new refactoring on the input metamodel, it can readily be applied on all target metamodels if adaptations are provided.

Although we show reuse of a kind of model transformations, namely refactorings, we claim its extensibility to other endogenous model transformations. In addition, our approach also fits well in the context of metamodel evolution. Indeed, all model transformations written 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 an old version to a new one. Finally, our approach can be seen as a potential framework for reusing arbitrary model transformations for arbitrary metamodels.

6 Related Work

Reuse in MDE has not been sufficiently investigated as compared to 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 operations across several input domains [20]. 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 [20]. Aspects [21] and open-classes [15] are powerful generic programming techniques for adapting programs by augmenting their behavior in existing classes [22,23]. Other languages that provide support for generic programming are Haskell and Scala [24]. The use of Haskell has been investigated [25] 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 [26] 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.* proposes an architecture called *Model Bus* that allows the interoperability of a wide range of modeling services [27]. The term '*modeling service*' defines an operation having models as inputs and outputs such as model editing, 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 [28,29] 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 [30]. 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 [31]. 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.

In this paper, we combine ideas from two recently published papers on metamodel pruning [5] and manual specification of generic model refactoring [7]. In [7] the authors present an approach to manually specify generic model transformations and in particular refactorings. A generic metamodel is manually specified and a generic transformation is written for the generic input metamodel. Other target input metamodels are then adapted to the generic metamodel to achieve genericity and reuse. This approach is not applicable to legacy model transformations where we do not use a generic metamodel but an existing and possible large input metamodel such as UML. Adapting a target input metamodel to this large metamodel to make it its subtype is a very tedious task. It sometimes requires several unnecessary adaptations as many of the concepts may not be used in the transformation. We deal with this problem via metamodel pruning [5] in our work to automatically obtain the effective input metamodel which plays the role of the generic metamodel. This automatic synthesis of the effective input metamodel extends the approach in [7] to legacy model transformation written for arbitrary input metamodels. It also helps drastically reduce the number of required adaptations via aspect-weaving and the time for type matching.

7 Conclusion

In this paper, we present an approach to make model transformations reusable across structurally different metamodels. This approach relies on metamodel pruning, model typing and a mechanism of adaptation based mainly on the weaving of aspects. We illustrate our approach with the Pull Up Method refactoring and validate it for three different refactorings (Encapsulate Field, Move Method, and Pull Up Method) for three different industrial metamodels (Java, MOF, and UML) in a concrete application. We demonstrate that our approach ensures a flexible reuse of model transformations. This approach seems to be generalizable 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 transformations adapted to several different metamodels, in particular industry standards such as Java, MOF, and UML.

References

1. Biggerstaff, T.J., Perlis, A.J.: Software Reusability Volume I: Concepts and Models. Volume I. ACM Press, Addison-Wesley, Reading, MA, USA (1989)
2. Mili, H., Mili, F., Mili, A.: Reusing software: Issues and research directions. IEEE Transactions of Software Engineering **21**(6) (1995) 528–562
3. Basili, V.R., Briand, L.C., Melo, W.L.: How reuse influences productivity in object-oriented systems. Communications of ACM **39**(10) (1996) 104–116
4. Blanc, X., Ramalho, F., Robin, J.: Metamodel reuse with mof. (2005) 661–675
5. Sen, S., Moha, N., Baudry, B., Jezequel, J.M.: Metamodel pruning. In: Model Driven Engineering Languages and Systems, 12th International Conference (MODELS), Denver, CO, USA (October 4-9 2009)

6. Steel, J., Jézéquel, J.M.: On model typing. Journal of Software and Systems Modeling (SoSyM) **6**(4) (December 2007) 401–414
7. Moha, N., Mahé, V., Barais, O., Jézéquel, J.M.: Generic model refactorings. (2009) 628–643
8. Fowler, M.: Refactoring – Improving the Design of Existing Code. 1st edn. Addison-Wesley (June 1999)
9. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: MODELS/UML. Volume 3713., Montego Bay, Jamaica, Springer (October 2005) 264–278
10. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science **152** (March 2006) 125–142
11. Janssens, D., Demeyer, S., Mens, T.: Case study: Simulation of a lan. Electronic Notes in Theoretical Computer Science **72**(4) (2003)
12. Hoffman, B., Pérez, J., Mens, T.: A case study for program refactoring. In: GraBaTs. (September 2008)
13. OMG: Mof 2.0 core specification. Technical Report formal/06-01-01, OMG (April 2006) OMG Available Specification.
14. OMG: The uml 2.1.2 infrastructure specification. Technical Report formal/2007-11-04, OMG (April 2007) OMG Available Specification.
15. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.D.: Multijava: Modular open classes and symmetric multiple dispatch for java. In: Proceedings of the 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). (2000) 130–145
16. OMG: The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07 (2007)
17. 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** (1999) 50–75
18. Steel, J.: Typage de modèles. PhD thesis, Université de Rennes 1 (April 2007)
19. Kermeta: http://www.kermeta.org/.
20. Gibbons, J., Jeuring, J., eds.: Generic Programming, IFIP TC2/WG2.1 Working Conference on Generic Programming, July 11-12, 2002, Dagstuhl, Germany. In Gibbons, J., Jeuring, J., eds.: Generic Programming. Volume 243 of IFIP Conference Proceedings., Kluwer Academic Publishers (2003)
21. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP). Volume 1241., Springer-Verlag (June 1997) 220–242
22. Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectj. SIGPLAN Not. **37**(11) (2002) 161–173
23. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: Proceedings of the 27th international conference on Software engineering (ICSE '05), New York, NY, USA, ACM (2005) 49–58
24. Oliveira, B.C.D.S., Gibbons, J.: Scala for generic programmers. In Hinze, R., Syme, D., eds.: WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming, New York, NY, USA, ACM (2008) 25–36

25. Lämmel, R.: Towards Generic Refactoring. In: Proceedings of Third ACM SIGPLAN Workshop on Rule-Based Programming RULE'02, Pittsburgh, USA, ACM Press (October5 2002) 14 pages.

26. et al., M.O.: An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)

27. Blanc, X., Gervais, M.P., Sriplakich, P.: Model bus : Towards the interoperability of modelling tools. In: European Workshop on Model Driven Architecture: Foundations and Applications (MDAFA'2004). Volume 3599 of LNCS., Springer (2004) 17–32

28. 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'08), Washington, DC, USA, IEEE Computer Society (2008) 211–218

29. Mnch, M.: Generic Modelling with Graph Rewriting Systems. PhD thesis, RWTH Aachen (2003) Berichte aus der Informatik.

30. 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: vol. 2: applications, languages, and tools. World Scientific Publishing, Singapore (1999) 669–689

31. Engels, G., Heckel, R., Cherchago, A.: Flexible interconnection of graph transformation modules. In: Formal Methods in Software and Systems Modeling. Volume 3393 of LNCS., Springer (2005) 38–63