Reusable model transformations

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

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

Reusable Model Transformations

Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, 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, barais, bbaudry, jezequel}e-mail: @irisa.fr

Received: date / Revised version: date

Abstract Model transformations written for an input metamodel may often 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 languages share concepts such as classes, methods, attributes, inheritance. Deriving motivation from this example, 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 analyze a transformation to obtain an effective subset of used concepts. We prune the input metamodel of the transformation to obtain an effective input metamodel containing the effective subset. The effective input metamodel represents the true input domain of transformation. (2) We adapt a target input metamodel by weaving it with aspects such as properties derived from the effective input metamodel to ultimately 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 without *any change* in the transformation itself. We validate our approach by adapting well-known refactoring transformations (Encapsulate Field, Move Method, and Pull Up Method) written for an in-house domain-specific modelling language (DSML) to three different industry standard metamodels (Java, MOF, and UML).

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

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 community [4]. One of the primary difficulties in making a model transformation reusable across different input domains is the difference in structural aspects between commutable/interchangeable input metamodels. Consider an example where model transformation reuse becomes obvious and yet is infeasible due to structural differences in commutable input metamodels. The example consists of a model transformation to *refactor models of class diagrams*, which is possible in several modelling languages supporting the concepts/types of classes, methods, attributes, and inheritance. For instance, the metamodels for the languages Java, MOF (Meta Object Facility), and UML all contain concepts/types needed to specify classes. If we emphasize the necessity for reuse then the refactoring transformation must be intuitively adaptable to all three metamodels as they manipulate similar models containing objects of similar types. Hence, we ask: How do we reuse one implementation of a model transformation for other type-theoretically similar modelling languages?

Our aim is to enable flexible reuse of model transformations across various type-theoretically similar metamodels to enhance productivity and modularity 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 transformation's input metamodel. We call the effective subset an *effective input metamodel* which represents the true input domain of the legacy model transformation. By definition in model type theory [5] the subtype property or the type conformance permits the legacy model transformation to process pertinent models conforming to the target input metamodel. Concisely,

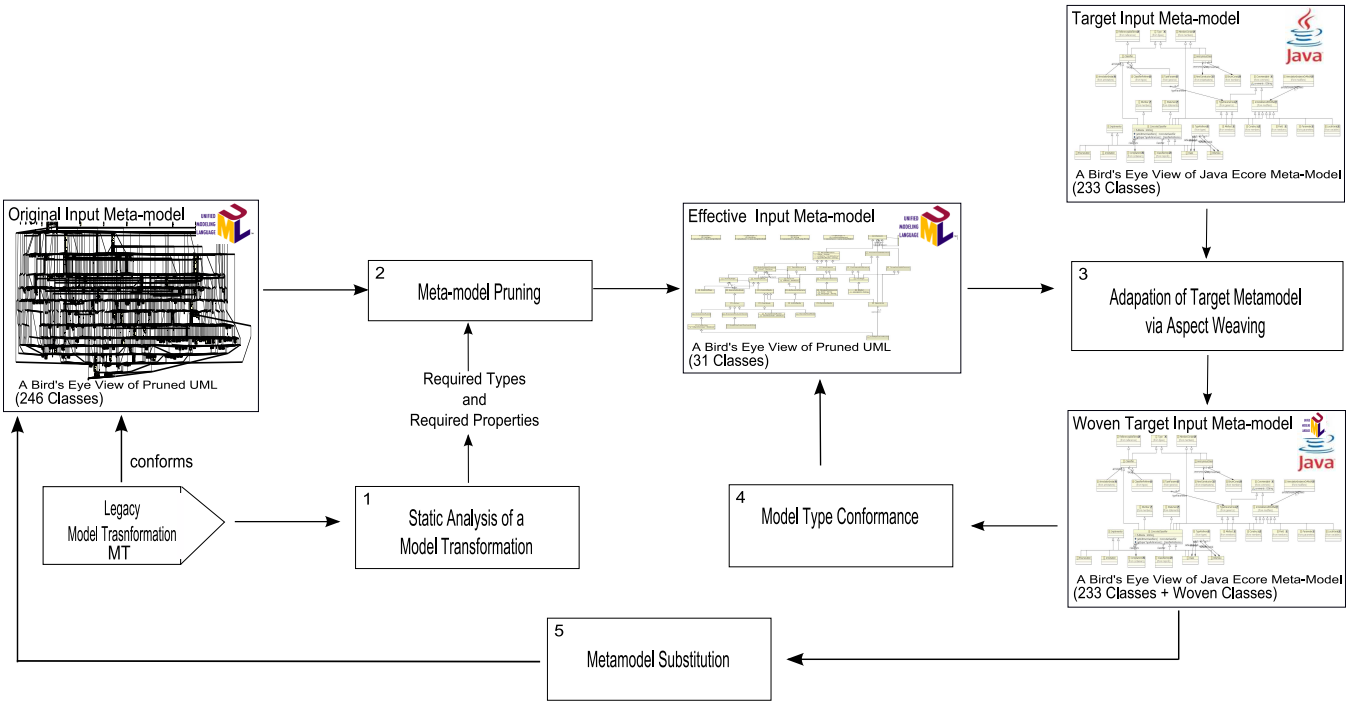

Figure 1 Overview of the Approach

our approach, depicted in Figure 1, follows these steps: (1) We perform a static analysis of the legacy transformation to extract the types and properties required in the transformation. (2) We automatically obtain an effective input metamodel via *metamodel pruning* [6] based on the types and properties required in the transformation. 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. (4) We adapt a target input metamodel by weaving it with structural aspects from the effective input metamodel. We also weave accessor functions for these structural aspects that seek information from related types in the target input metamodel. For example, if Java is the target input metamodel and UML is the original input metamodel for a legacy transformation then values from properties in Java input models must stay synchronized with UML properties actually handled by the transformation. Moreover, the Java input models must temporarily (during execution) contain properties derived from UML that are identifiable by the transformation for UML models. These identifiable properties in fact are part of the effective input metamodel. Therefore, the woven aspects are derived properties and accessor functions that help make the Java metamodel a subtype of UML. (3) We use *model typing* [7] 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. Our approach is infeasible when the target input metamodel cannot be adapted to show type conformance with the effective input metamodel for some type(s) used in the transformation. (5) Replacing the original input metamodel with the woven target input metamodel at execution allows the legacy model transformation to process relevant input models conforming to the target input metamodel.

The scientific contribution of our approach is based on a combination of two recent ideas; namely, metamodel pruning [6] and manual specification of generic model refactorings [8]. In [8], the authors manually specify a generic model transformation for a hand-made generic metamodel that is adapted to various target input metamodels. The generic model transformation performs refactoring on models instances of the generic metamodel and all metamodels adapted to the generic metamodel. The generic metamodel, presented in [8], is a lightweight metamodel that contains a minimum set of concepts (such as classes, methods, attributes, and parameters) common to various class diagram like metamodels such as Java and UML. In our work, we automatically synthesize an effective input metamodel via metamodel pruning [6], which is in contrast to manually specifying a generic metamodel as done in [8]. 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 [8]. The adaptation of target input metamodels to the effective input metamodel via aspect weaving remains similar to the approach in [8].

We demonstrate our approach on well-known model transformations; namely, refactorings [9]. A refactoring is a particular transformation performed on the struc-

ture of software to make it easier to understand and modify without changing its observable behavior [9]. For example, the refactoring Pull Up Method consists of moving methods to the superclass if these methods have the same signatures and/or results on subclasses [9]. 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 the transformation language Kermeta [10].

This article is organized as follows. In Section 2, we describe motivating examples that illustrate the key challenges. In Section 3, we introduce foundations necessary to describe our approach. The foundations include a description of the executable metamodeling language, Kermeta, highlighting some of its new features including the notion of model typing, and a presentation of metamodel pruning to obtain an effective input metamodel. Section 4 gives a general step-by-step overview of our approach. 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 8 concludes and presents future work.

2 Motivating Example

Let us suppose that a company needs to economically upgrade its legacy transformations from an old input metamodel to a new but similar industry standard metamodel such as the latest UML. The old metamodel may either be from an in-house DSML or an old version of an industry standard such as UML. The legacy transformation itself must remain unchanged.

Let us now consider an example of a model transformation that can refactor the in-house DSML. The DSML itself is used to model 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 [9]. 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 [11] of a Local Area Network (LAN) application [12] and adapted in Figure 2 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 [13] represents Java programs with some restrictions over the Java code. For example, inner classes, anonymous classes, and generic types are not modeled. As a MOF metamodel, we consider the metamodel of Kermeta [10], which is an extension of MOF [14] with an imperative action language for specifying constraints and operational semantics of metamodels. The UML metamodel studied in this paper corresponds to version 2.1.2 of the UML specification [15]. This Java metamodel is *one* possible specification for Java programs; we may use another Java metamodel based on the specification of the Abstract Syntax Tree Metamodel (ASTM) provided by the OMG ADM (Architecture-Driven Modernization) group [16].

We provide an excerpt of each of these metamodels in Figures 3, 4, and 5. 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 encounter several problems if we intend to specify a common Pull Up Method refactoring for all three metamodels:

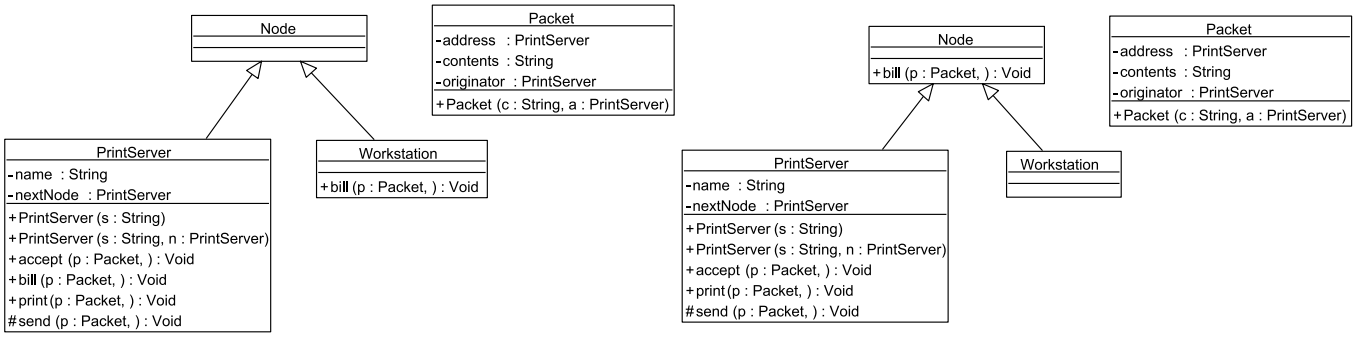

Figure 2 Class Diagrams of the LAN Application Before and After the Pull Up Method Refactoring of the Method `bill`.

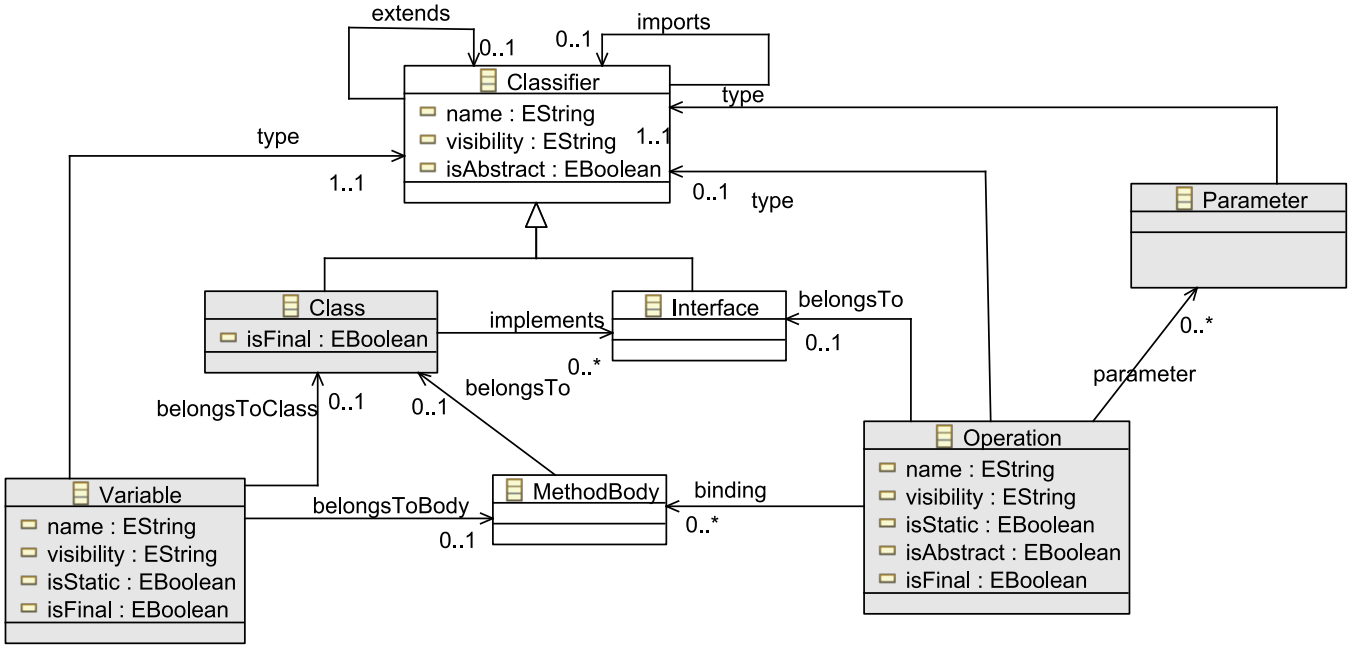

Figure 3 Subset of the Java Metamodel.

- **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 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 our 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.

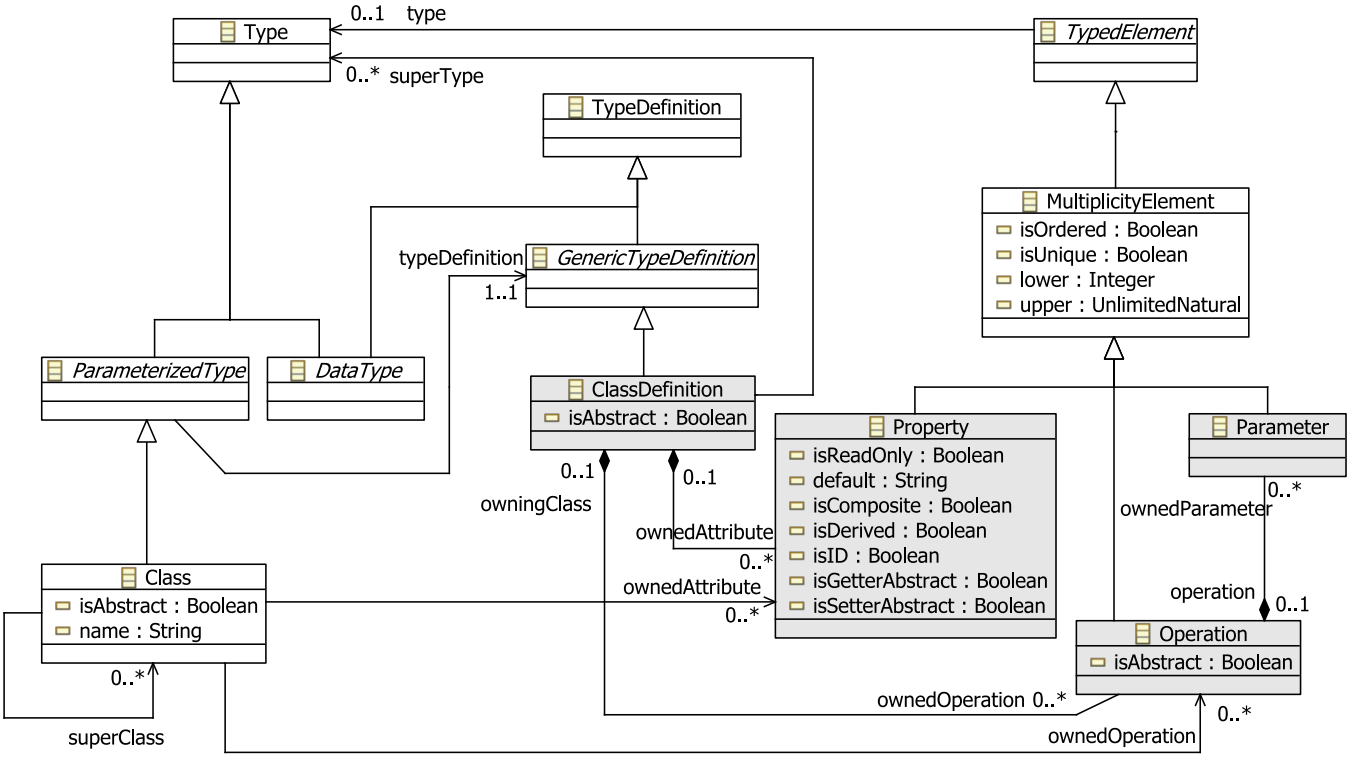

Figure 4 Subset of the MOF Metamodel.

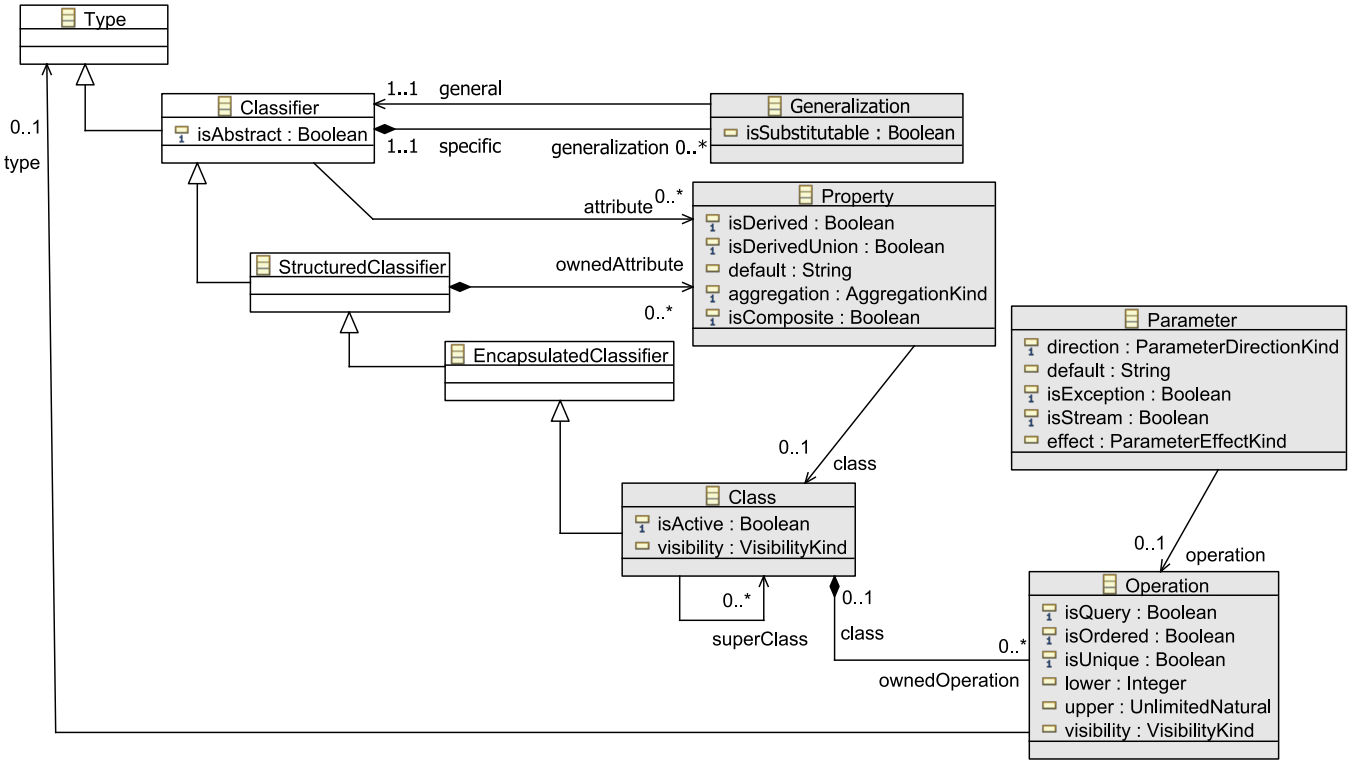

Figure 5 Subset of the UML Metamodel.

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 relevant Kermeta features 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 MOF standard [14]. 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 [10]. Kermeta is built on top of Eclipse Modeling Framework (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 Features of Kermeta

The action language of Kermeta provides some features for weaving aspects, adding derived properties, and specifying constraints such as invariants and pre-/post-conditions. Indeed, the first 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 [17]). Aspect weaving consists of composing a base model with aspects defining new concerns, thereby yielding a base model with new structure and behavior. This feature offers more flexibility to developers by enabling them to easily manipulate and reuse existing metamodels while separating concerns. The second 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. The feature amounts to the possibility of determining the value of a property based on the values of other properties. These other properties may come from the same class and/or from properties reachable through the navigation of the metamodel. The last pertinent Kermeta 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 [18].

3.3 Model Typing

The Kermeta language integrates the notion of model typing [7], 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 [19]. 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 [7] and improved here by relaxing two strong constraints. First, the constraint related to the name-dependent conformance on properties and operations was relaxed by enabling their renaming. The second constraint related to the strict structural conformance was relaxed by extending the matching to subclasses.

Let's illustrate model typing with two metamodels *M* and *M'* given in Figures 6 and 7. 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 [5]. The interested

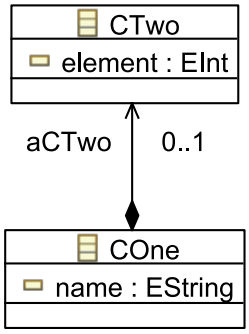

Figure 6 Metamodel *M*.

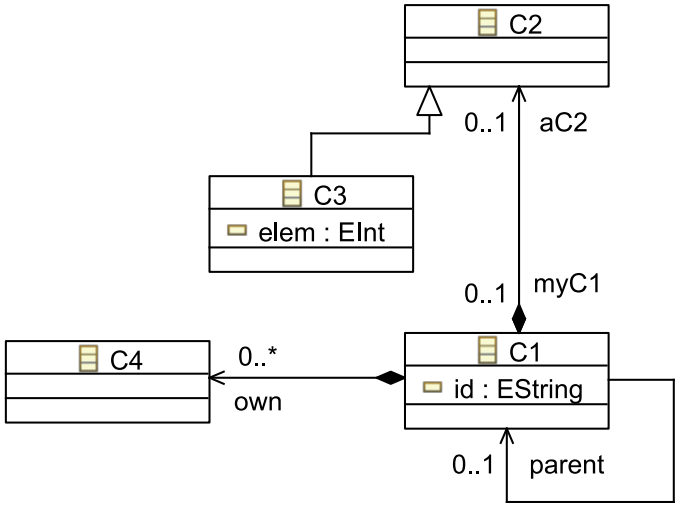

Figure 7 Metamodel *M'*.

reader can find in [5] the details of matching rules used for model types.

However, 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

Metamodel pruning [6] is an algorithm that outputs an effective subset metamodel of a possible large input metamodel such as the UML. The output effective metamodel conserves a set of required types and properties (given as input to metamodel pruning) and all its obligatory dependencies (computed by the algorithm). The algorithm prunes every other type and property. In the type-theoretic sense the resulting effective metamodel is a *supertype* of the large input metamodel. We verify the supertype property using model typing [7]. We concisely describe the metamodel pruning algorithm in the following paragraphs.

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 diagrams. 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 required properties P_{req} and prune away the rest in a metamodel. The authors of [6] 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 model transformations 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 subclasses 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 [6] 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 8 and describe the steps in the approach below:

Step 1: Static Analysis of a Transformation

As shown in Figure 8 we first perform *static analysis* on the legacy model transformation `MT`. Static analysis can be extrapolated to several model transformations when they are called and navigable from a main transformation. The main transformation is given as an input

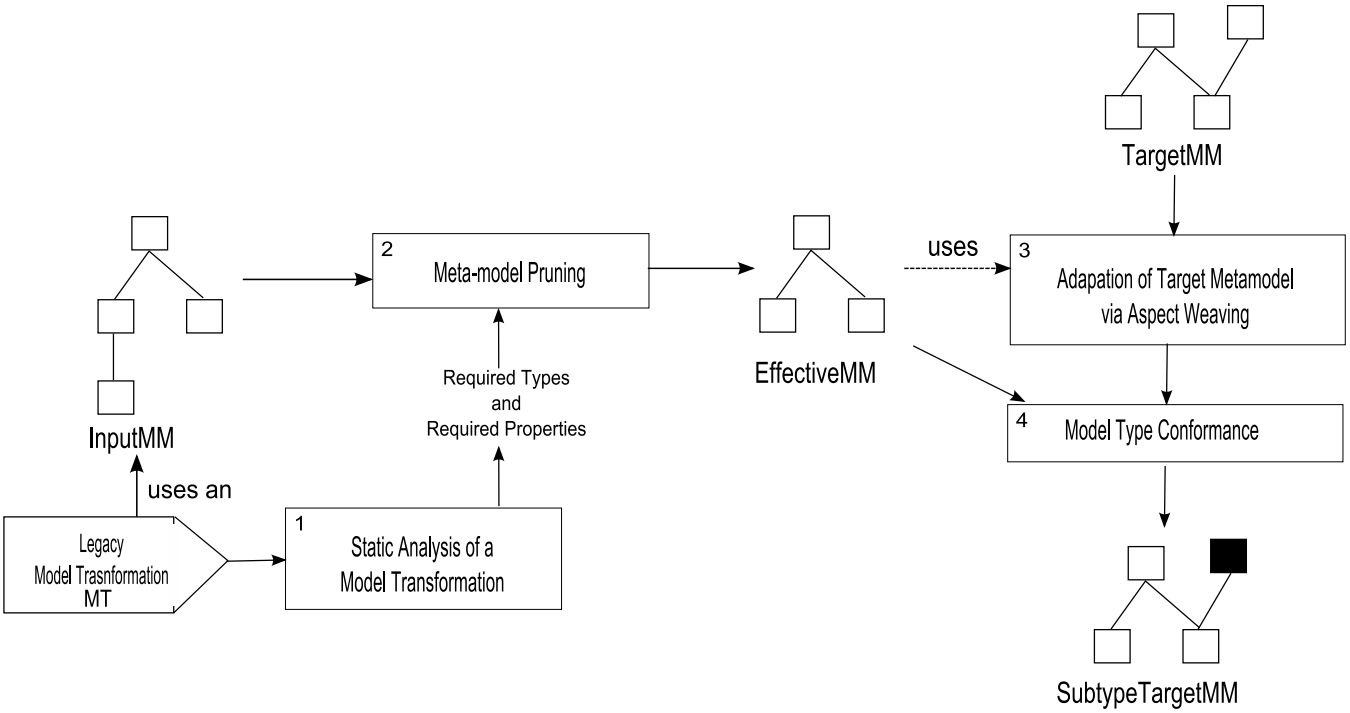

Figure 8 Approach for Transforming an Input Metamodel to Subtype Target Input Metamodel

to the static analysis process. The static 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. The static analysis can only be performed when the source code for the transformation is available. If not, the required types and properties must be manually specified. 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 input metamodel must be compared with the target metamodel.

The metamodel pruning process plays a key role when the input domain of a transformation corresponds to a 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, becomes very tedious unless only a subset of the input metamodel is in use.

Step 3: Aspect Weaving of Target Metamodel

One of the new features of Kermeta is to weave aspects (see Section 3.2) 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 recalled as the subtype target in-

put 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 9.

5 Experiments and Discussion

We performed some experiments by applying our approach to legacy model refactoring transformations (Encapsulate Field, Move Method, and Pull Up Method [9]) written for an in-house DSML to three target industry standard input metamodels Java, UML, and MOF. Our goal is to be able to reuse these three well known refactorings on models of the LAN application [12] conforming to the three different metamodels.

We illustrate our approach using the specific example of the Pull Up Method refactoring transformation. The implementation of the example is in Listing 1 (see Appendix A.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.

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 a static analysis of refactoring model transformations (Encapsulate Field, Move Method, and Pull Up Method) applied on an in-house DSML for the INRIA team TRISKELL from Rennes, France. 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. This drastically reduces the number of adaptations required in the target input metamodels: Java, MOF, and UML. The DSML contains several other classes related to contracts and verification. 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.

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 10. 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 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. We weave missing types, properties and their opposite properties from the EffectiveMM into the TargetMM. These properties include *getter* and *setter* accessors that seek information in the TargetMM to assign values to the derived properties woven from EffectiveMM. This step of adaptation is necessary because model typing is too restrictive for allowing a matching between metamodels that are structurally very 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 $SubtypeTargetMM$, as seen in Figure 9.

To better understand the adaptation process we illustrate it with a simple example shown in Figure 11. In Figure 11 (a), type Class exists in the effective input metamodel EffectiveMM and Classifier exists in the target input metamodel TargetMM Java. We ask in Figure 11 (b) if the the types match with respect to model typing rules and the answer is *no* because the properties superClasses and subClasses do not appear in TargetMM. Hence, we weave the properties superClasses and subClasses from Class into Classifier as shown in Figure 11 (c). These properties are computed using the already existing property extends in TargetMM. Now, the types Class and Classifier match as seen in Figure 11 (d). This process is repeated for *every type* in EffectiveMM such that a conforming type is created in the target input metamodel. If a match for a type is not found or multiple matches for a type in the EffectiveMM are found then the target input metamodel is unadaptable and our approach fails.

In the following paragraphs, we describe technical details of the adaptations for the target input metamodels Java, MOF, and UML such that they type conform with the effective input metamodel EffectiveMM of refactoring transformations. In particular, we describe the adaptations of the derived properties superClasses and subClasses of Class for the target input metamodels. We discuss only the woven getter accessors of the derived properties; the setter accessors are symmetric.

Adaptation for the Java metamodel. The Listing 2 in Appendix A.2 describes the adaptation made to the Java

[1] The interested reader can refer to the Kermeta syntax in [20].

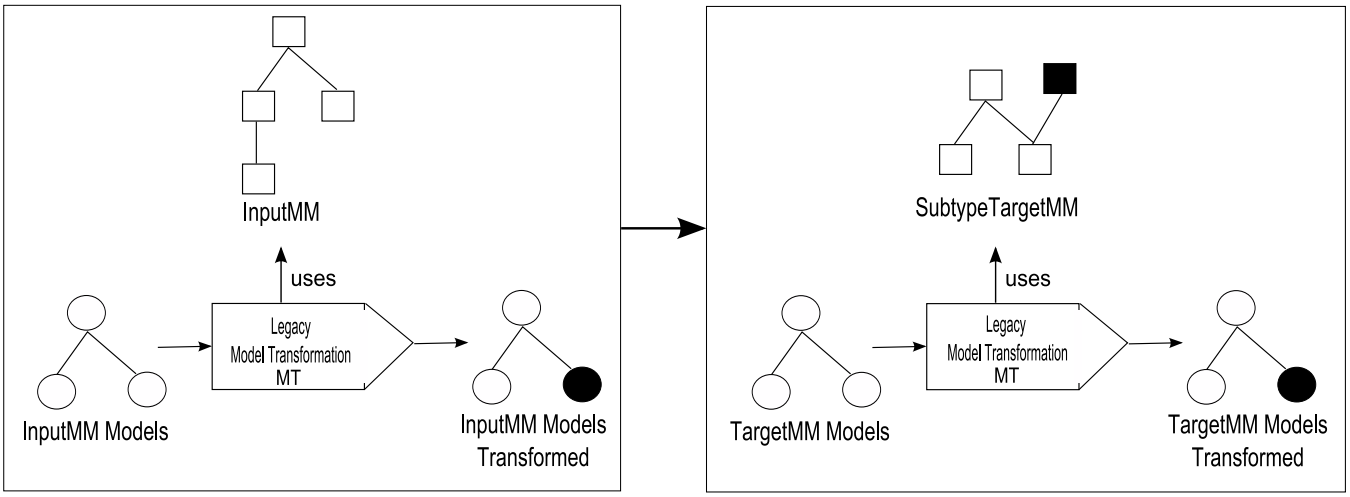

Figure 9 The Legacy Transformation used as a Generic Transformation for TargetMM

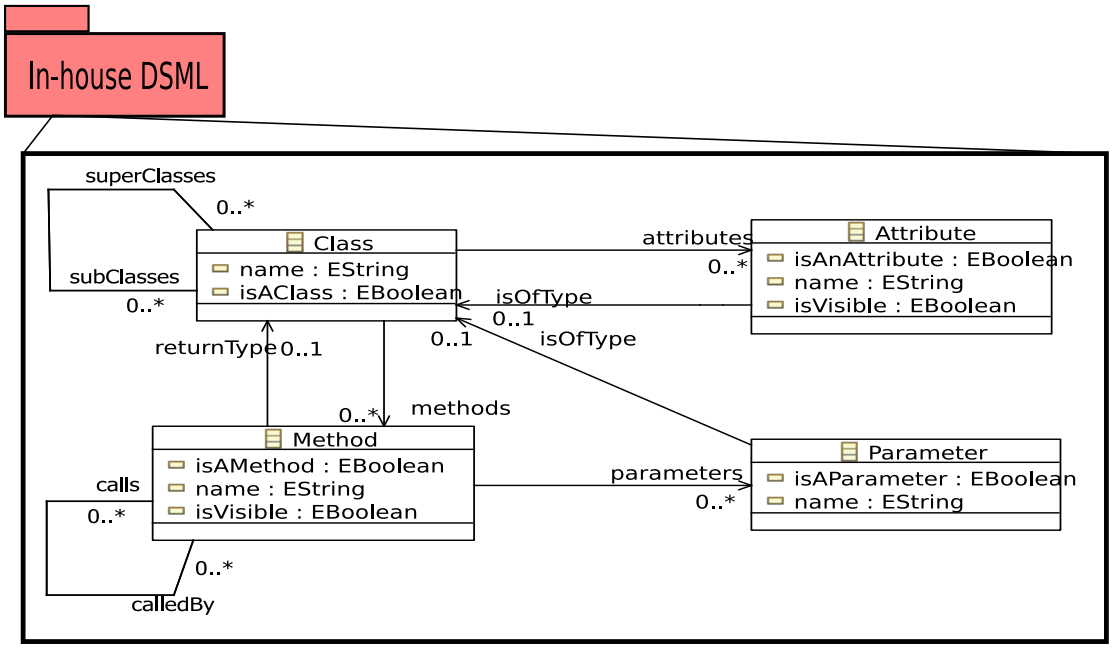

Figure 10 Effective Metamodel `EffectiveMM` extracted from an In-house DSML via Pruning

metamodel to adapt it to `EffectiveMM`. The adaptation is applied on a subset of the Java metamodel shown in Figure 3. The derived property `superClasses` corresponds to a simple access to the property `extends` that is then wrapped in a Java `Class` (Lines 12–15). However, for the derived property `subClasses`, the opposite `inv_extends` of the property `extends` was weaved by an aspect on the class `Classifier` and used to get the set of subclasses (Lines 17–21).

Adaptation for the MOF metamodel. The Listing 3 in Appendix A.3 describes the adaptation made to the MOF metamodel to adapt it to `EffectiveMM`. We apply the adaptation on a subset of the MOF metamodel shown in Figure 4. 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 3 (Lines 5–15).

Adaptation for the UML metamodel. The Listing 4 in Appendix A.4 describes the adaptation made to the UML metamodel to adapt it to `EffectiveMM`. We apply the adaptation on a subset of the UML metamodel shown in Figure 5. In UML, the inheritance links are reified through the class `Generalization` (Lines 5–7). Thus, the derived property `superClasses` is computed by accessing the class `Generalization` and the reference property `general` (Lines 11–15). As in Java and MOF, an opposite `inv_general` is specified to get the set of subclasses (Lines 17–21).

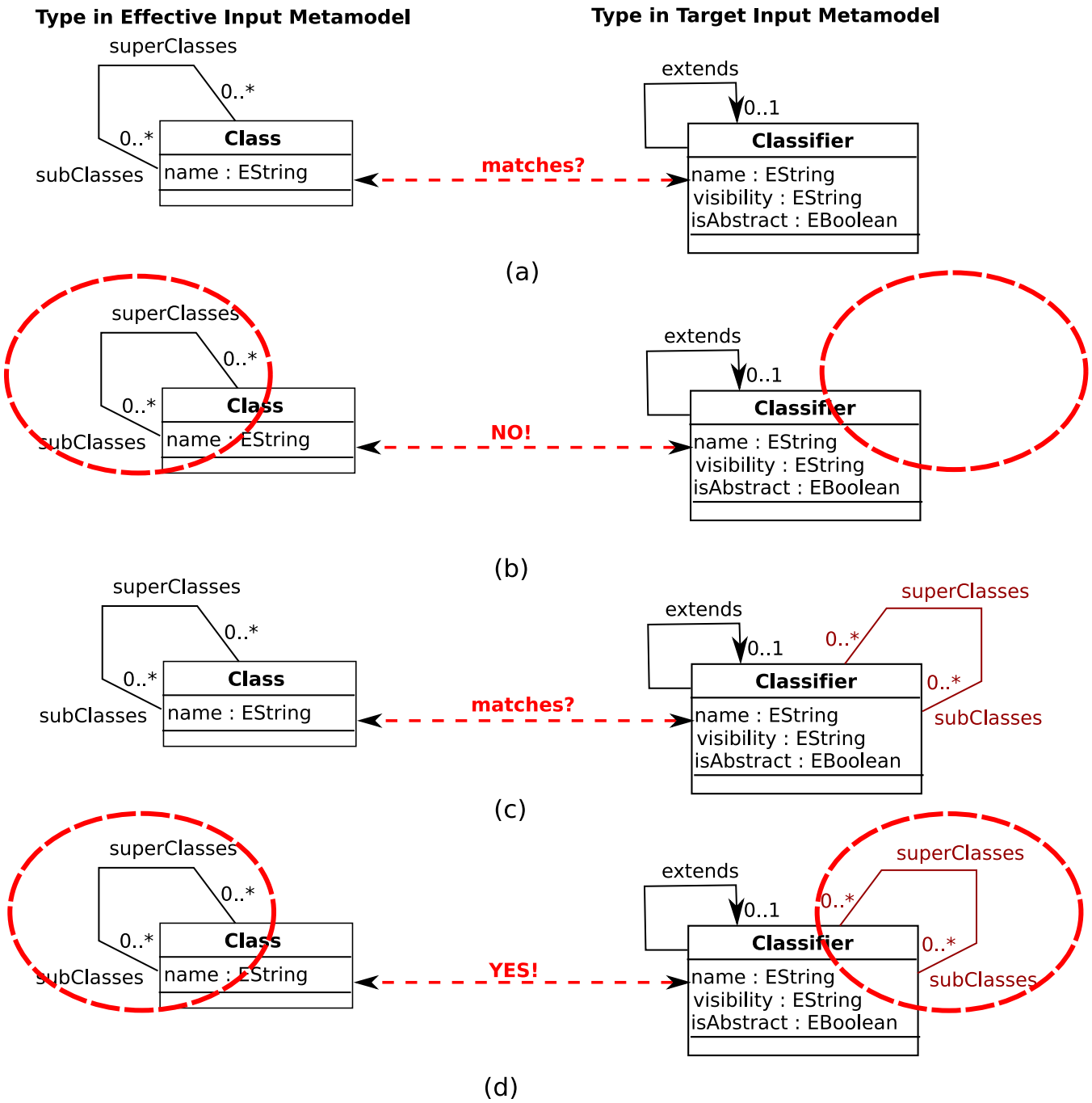

Figure 11 An Example of Weaving Steps for Adaptation

Finally, we apply the refactoring on the target input metamodels as illustrated in Listing 5 (see Appendix A.5) for the UML metamodel. We reuse the example of the method `bill` in the LAN application (Lines 12, 16). We notice that the class `Refactor` takes as an argument the UML metamodel (Lines 18–19), which due to the adaptation of Listing 4 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 `EffectiveMM`.

5.2 Discussion

We also experimented with a fourth metamodel as shown in Figure 12. In this metamodel, the two classes (corresponding to `Class` and `Parameter` in the effective input metamodel) are unified in the 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

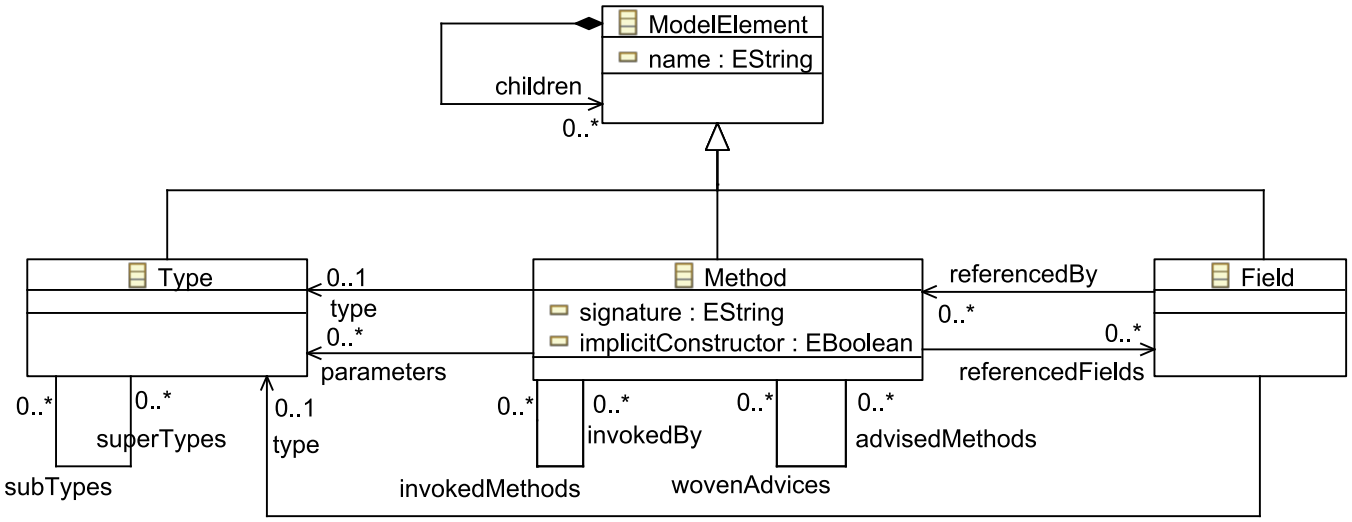

Figure 12 Subset of the Fourth Metamodel.

metamodel should correspond to a distinct element in the target input metamodel. We specify a fourth refactoring, Extract Method [9], that creates a new method from a code fragment. The Extract Method refactoring uses the concept of *method body* but this concept is missing in the UML metamodel. Therefore, the missing concept prevents the ability to reuse the refactoring for the UML metamodel. However, during the adaptation step, we could fill in this difference by weaving the missing concept to the UML metamodel as an aspect. Our approach is thus not very restrictive since the mechanism of adaptation enables the raising of awareness of inherent limitations.

In [21], we apply some refactorings on a Java metamodel with a flat structure (*i.e.*, with no containers). The search for elements in such a metamodel is not optimal since we need to traverse all elements in the flat structure. However, in the current paper, the navigation of the elements is easier thanks to opposites properties. These properties enable bi-directional traversal of a metamodel. The addition of opposites is done automatically while loading metamodels in the Kermeta platform.

Our approach theoretically relies on the model typing and is feasible in practice thanks to the mechanism of metamodel pruning and aspect weaving based adaptation. Writing adaptations can be more or less difficult depending on the developers' knowledge of the target input metamodels. Our approach is relevant if the number of transformations to be reused is significant. This means that the effort to convert the transformations is greater than writing adaptations of a metamodel. 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 predict its extensibility to arbitrary model transformations with arbitrary input metamodels. 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.

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 adaptable using generic operations that are functional across several input domains [22]. 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 [22]. Aspects [23] and open-classes [17] are powerful generic programming techniques for adapting programs by augmenting their behavior in existing classes [24,25]. Other languages that provide support for generic programming are Haskell and Scala [26]. The use of Haskell has been investigated [27] 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 [28] 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 seamless integration with Java. However, it would require writing a significant amount of code and manage relationships among generic types.

In the MDE community, Blanc *et al.* propose an architecture called *Model Bus* that allows the interoperability of a wide range of modeling services [29]. 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 works [30,31] 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 [32]. 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 [33]. 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.

7 Discussion

In this paper, we combine ideas from two recently published papers on metamodel pruning [6] and manual specification of generic model refactoring [8]. In [8], 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 [6] 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 [8] 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. Adaptation followed by verification using model typing, used in our approach, may be compared to generic pattern-matching techniques [34, 35]. These pattern matching techniques can automatically detect concept similarities between metamodels. However, these similarities are often limited to the syntax of the metamodels and not their intended semantics. For instance, the notion of a Class may have very different meanings in two metamodels. Simply matching the concept Class and its structure in two metamodels does not ascertain a 100% conformance between these seemingly similar types. A number of ambiguities may crop up due to same name but different structure of concepts while pattern matching. Human intervention is required to clearly build a bridge between concepts in two metamodels. The precise mechanism of adapting a metamodel using aspects followed by verification using model typing overcomes the limitations of classical pattern-matching mechanisms.

8 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. We enlist the limitations of our approach based on the theoretical foundations of model typing [5]. We predict that our approach could be generalizable to arbitrary model transformations that can be used for various input domains such as the computation of metrics, detection of patterns, and inconsistencies. As future work, we plan to increase the repository of legacy transformations adapted to several different metamodels, in particular industry standards such as Java, MOF, and UML. We intend to apply our approach to the reuse of OCL constraints used as pre-/post-conditions of model transformations.

A Appendix

A.1 Kermeta Code for the Pull Up Method Refactoring.

```
1 package refactor;
2
3 class Refactor<MT : EffectiveMM> {
4
5   operation pullUpMethod(
6                 source : MT::Class,
7                 target : MT::Class,
8                 meth   : MT::Method) : Void
9
```

```
10    // Preconditions
11    pre sameSignatureInOtherSubclasses is do
12          target.subClasses.forAll{ sub |
13              sub.methods.exists{ op |
                    haveSameSignature(meth, op) } }
14    end
15
16    // Operation body
17    is do
18        target.methods.add(meth)
19        source.methods.remove(meth)
20    end
21  }
```

Listing 1 Kermeta Code for the Pull Up Method Refactoring.

A.2 Kermeta Code for Adapting the Java Metamodel.

```
1  package java;
2
3  require "Java.ecore"
4
5  aspect class Classifier {
6    reference inv_extends : Classifier[0..*]#extends
7    reference extends : Classifier[0..1]#inv_extends
8  }
9
10 aspect class Class {
11
12   property superClasses : Class[0..1]#subClasses
13     getter is do
14       result:=self.extends
15     end
16
17   property subClasses : Class[0..*]#superClasses
18     getter is do
19       result := OrderedSet<java::Class>.new
20       self.inv_extends.each{subC|result.add(subC)}
21     end
22 }
```

Listing 2 Kermeta Code for Adapting the Java Metamodel.

A.3 Kermeta Code for Adapting the MOF Metamodel

```
1  package kermeta;
2
3  require kermeta
4
5  aspect class ParameterizedType {
6   reference typeDefinition: GenericTypeDefinition
        [1..1]#inv_typeDefinition
7  }
8
9  aspect class GenericTypeDefinition {
10  reference inv_typeDefinition: ParameterizedType
        [1..1]#typeDefinition
11 }
12
13 aspect class Type {
14  reference inv_superType: ClassDefinition[0..*]#
        superType
15 }
16
17 aspect class ClassDefinition {
18
19  reference superType : Type[0..*]#inv_superType
20
21  property superClasses : ClassDefinition[0..*]#
        subClasses
22   getter is do
23     result := OrderedSet<ClassDefinition>.new
24     self.superType.each{ c |
25       var clazz : Class init Class.new
26       clazz ?= c
27       var clazzDef : ClassDefinition init
             ClassDefinition.new
28       clazzDef ?= clazz.typeDefinition
29       result.add(clazzDef) }
30   end
31
32  property subClasses : ClassDefinition[0..*]#
        superClasses
33   getter is do
34     result := OrderedSet<ClassDefinition>.new
35     var clazz : Class
36     clazz ?= self.inv_typeDefinition
37     clazz.inv_superType.each{ superC | result.add(
          superC) }
38   end
39 }
```

Listing 3 Kermeta Code for Adapting the MOF Metamodel.

A.4 Kermeta Code for Adapting the UML Metamodel

```
1  package uml;
2
3  require "http://www.eclipse.org/uml2/2.1.2/UML"
4
5  aspect class Classifier {
6   reference inv_general : Generalization[0..*]#
        general
7  }
8
9  aspect class Class {
10
11  property superClasses : Class[0..*]#subClasses
12   getter is do
13     result := OrderedSet<uml::Class>.new
14     self.generalization.each{ g | result.add(g.
          general) }
15   end
16
17  property subClasses : Class[0..*]#superClasses
18   getter is do
19     result := OrderedSet<uml::Class>.new
20     self.inv_general.each{ g | result.add(g.
          specific) }
21   end
22 }
```

Listing 4 Kermeta Code for Adapting the UML Metamodel.

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

```
1  package refactor;
2
3  require "http://www.eclipse.org/uml2/2.1.2/UML"
4
5  class Main {
6
7   operation main() : Void is do
8
9    var rep : EMFRepository init EMFRepository.new
10
11   var model : uml::Model
12   model ?= rep.getResource("lan_appli.uml").one
13
14   var source: uml::Class init getClass("
         PrintServer")
15   var target: uml::Class init getClass("Node")
16   var meth  : uml::Operation init getOperation("
         bill")
17
```

```
18   var refactor : refactor::Refactor<uml::UmlMM>
19           init refactor::Refactor<uml::UmlMM>.new
20
21   refactor.pullUpMethod(source, target, meth)
22  end
23 }
```

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

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. In: ACM/IEEE 8^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS'05). (Oct 2005) 661–675
5. Steel, J.: Typage de modèles. PhD thesis, Université de Rennes 1 (April 2007)
6. Sen, S., Moha, N., Baudry, B., Jezequel, J.M.: Metamodel pruning. In: ACM/IEEE 12^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS'09), Springer (Oct 2009) 32–46
7. Steel, J., Jézéquel, J.M.: On model typing. Journal of Software and Systems Modeling (SoSyM) **6**(4) (December 2007) 401–414
8. Moha, N., Mahé, V., Barais, O., Jézéquel, J.M.: Generic Model Refactorings. In: ACM/IEEE 12^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS'09), Springer (Oct 2009) 628–643
9. Fowler, M.: Refactoring – Improving the Design of Existing Code. 1^{st} edn. Addison-Wesley (June 1999)
10. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: ACM/IEEE 8^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS'05), Springer (Oct 2005) 264–278
11. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science **152** (March 2006) 125–142
12. Janssens, D., Demeyer, S., Mens, T.: Case study: Simulation of a LAN. Electronic Notes in Theoretical Computer Science **72**(4) (2003)
13. Hoffman, B., Pérez, J., Mens, T.: A case study for program refactoring. In: 4^{th} International Workshop on Graph-Based Tools (GraBaTs'08). (September 2008)
14. OMG: MOF 2.0 core specification. Technical Report formal/06-01-01, OMG (April 2006)
15. OMG: The UML 2.1.2 infrastructure specification. Technical Report formal/2007-11-04, OMG (April 2007)
16. OMG: Architecture-driven modernization (ADM): Abstract syntax tree metamodel (ASTM) 1.0 - beta 2. Technical Report ptc/09-07-06, OMG (July 2009)
17. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.D.: Multijava: Modular open classes and symmetric multiple dispatch for java. In: 15^{th} ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00). (October 2000) 130–145
18. OMG: The object constraint language specification 2.0, OMG. Technical Report ad/03-01-07, OMG (January 2003)
19. 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
20. Kermeta: http://www.kermeta.org/ Accessed on April 2010.
21. Moha, N., Sen, S., Faucher, C., Barais, O., Jézéquel, J.M.: Evaluation of kermeta on graph transformation problems. International Journal on Software Tools for Technology Transfer (STTT) (2010) To appear
22. Gibbons, J., Jeuring, J., eds.: Generic Programming. (July)
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: 11^{th} European Conference on Object-Oriented Programming (ECOOP'97). Volume 1241., Springer-Verlag (June 1997) 220–242
24. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. SIGPLAN Not. **37**(11) (2002) 161–173
25. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: 27^{th} International Conference on Software Engineering (ICSE'05). (2005) 49–58
26. Oliveira, B., Gibbons, J.: Scala for generic programmers. In Hinze, R., Syme, D., eds.: ACM SIGPLAN Workshop on Generic Programming (WGP'08). (2008) 25–36
27. Lämmel, R.: Towards generic refactoring. In: 3^{rd} ACM SIGPLAN Workshop on Rule-Based Programming (RULE'02), ACM (October 2002) 15–28
28. Odersky, M., et al.: An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland (2004)
29. 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'04). Volume 3599 of LNCS., Springer (2004) 17–32
30. Amelunxen, C., Legros, E., Schurr, A.: Generic and reflective graph transformations for the checking and enforcement of modeling guidelines. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'08), IEEE Computer Society (2008) 211–218
31. Münch, M.: Generic Modelling with Graph Rewriting Systems. PhD thesis, RWTH Aachen (2003) Berichte aus der Informatik.
32. 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 Co., Inc. (1999) 639–689
33. Engels, G., Heckel, R., Cherchago, A.: Flexible interconnection of graph transformation modules. In: For-

mal Methods in Software and Systems Modeling. Volume 3393 of LNCS., Springer (2005) 38–63
34. Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., Jézéquel, J.M.: Introducing variability into aspect-oriented modeling approaches. In: ACM/IEEE 10^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS'07). LNCS, Springer (October 2007) 498–513
35. Ramos, R., Barais, O., Jézéquel, J.M.: Matching model-snippets. In: ACM/IEEE 10^{th} International Conference on Model Driven Engineering Languages and Systems (MODELS'07). LNCS, Springer (October 2007) 121–135