Theory and Practice of Model Transformations: 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings. Lecture Notes in Computer Science. Volumen 8568. Springer 2014. 186-201

DOI: http://dx.doi.org/10.1007/978-3-319-08789-4_14

Reverse Engineering of Model Transformations for Reusability

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara

Modelling and Software Engineering Research Group (`http://www.miso.es`)
Universidad Autónoma de Madrid (Spain)

Abstract. Reuse techniques are key for the industrial adoption of Model-Driven Engineering (MDE). However, while reusability has been successfully applied to programming languages, its use is scarce in MDE and, in particular, in model transformations.

In previous works, we developed an approach that enables the reuse of model transformations for different meta-models. This is achieved by defining reusable components that encapsulate a generic *transformation template* and expose an interface called *concept* declaring the structural requirements that any meta-model using the component should fulfil. Binding the concept to one of such meta-models induces an adaptation of the template, which becomes applicable to the meta-model. To facilitate reuse, concepts need to be concise, reflecting only the minimal set of requirements demanded by the transformation.

In this paper, we automate the reverse engineering of existing transformations into reusable transformation components. To make a transformation reusable, we use the information obtained from its static analysis to derive a concept that is minimal with respect to the transformation and maximizes its reuse opportunities, and then evolve the transformation accordingly. The paper describes a prototype implementation and an evaluation using transformations from the ATL zoo.

Keywords: Model transformation, Reusability, Reverse engineering, Re-engineering

1 Introduction

Reusability is a key enabler for the industrial adoption of Model-Driven Engineering (MDE). Some techniques have been proposed to reuse complete transformations, such as superimposition [19], phases [14] and genericity [13], but their use is still an exception. As noted by [1], one reason for this situation is the lack of repositories for selecting and effectively reusing transformations. Even the ATL Transformation Zoo [2], which is the closest relative to a transformation repository, consists of a collection of transformations not designed for reuse. This contrasts with the rich ecosystems of libraries in e.g., object-oriented languages like Java or C#, which successfully promote development with reuse.

In previous works [13], we proposed a technique for transformation reuse based on generic programming. In our approach, reusable transformation components encapsulate a transformation template developed against so-called concepts, which resemble meta-models but their elements are variables. Binding these variables to concrete meta-model elements induces a rewriting of the template to make it compatible with the meta-model. Thus, we obtain reusability because the transformation component can be used with any meta-model that can be bound to its concepts. However, this technique implies developing transformations with reusability up-front, by designing suitable concepts for the input and output domains and then writing the transformation template accordingly. Thus, it is not possible to profit from existing transformations beyond their use as a reference to manually implement a generic, reusable transformation. While concepts need to be concise to facilitate reuse and include only the elements accessed by a template, transformations are developed for concrete meta-models (e.g. UML) which reflect the complexity of a domain and may include accidental complexity from the transformation point of view. Hence, making an existing transformation reusable requires both a simplification of the meta-model into a truly reusable concept, and an according reorganization of the transformation.

In this work, we propose a semi-automatic process to reverse engineer existing transformations into generic, reusable transformations. It has been implemented for ATL as this is one of the most widely used transformation languages. Our aim is to foster reuse by facilitating the transition from existing, non-reusable transformations into reusable components that can be offered as transformation libraries in a repository. The process starts by extracting the effective meta-model of a transformation, which implies its static analysis to derive typing information. Then, the effective meta-model is evolved towards a concise concept through a series of refactorings, and the transformation is co-evolved accordingly if needed. The approach is supported by a prototype tool, and has been evaluated using transformations of the ATL zoo.

Organization. Section 2 presents our previous work on reusable transformations. Then, Section 3 overviews our proposal to the reverse engineering of existing transformations into reusable components, which is detailed in the following two sections: static analysis of ATL transformations (Section 4), and extraction and customization of concepts (Section 5). We evaluate our approach in Section 6, review related work in Section 7, and draw conclusions in Section 8.

2 Reusable transformations

In order to build a reusable transformation, in previous work [13] we proposed the notion of *transformation components* with a well-defined interface called *concept*. Fig. 1 shows a generic transformation component to calculate metrics for object-oriented languages, as well as its instantiation for a specific meta-model. The component (label 1) includes a transformation template from a hand-made concept characterising object-oriented languages to a metrics meta-model. We only show an excerpt of the template, which calculates the Depth of Inheritance

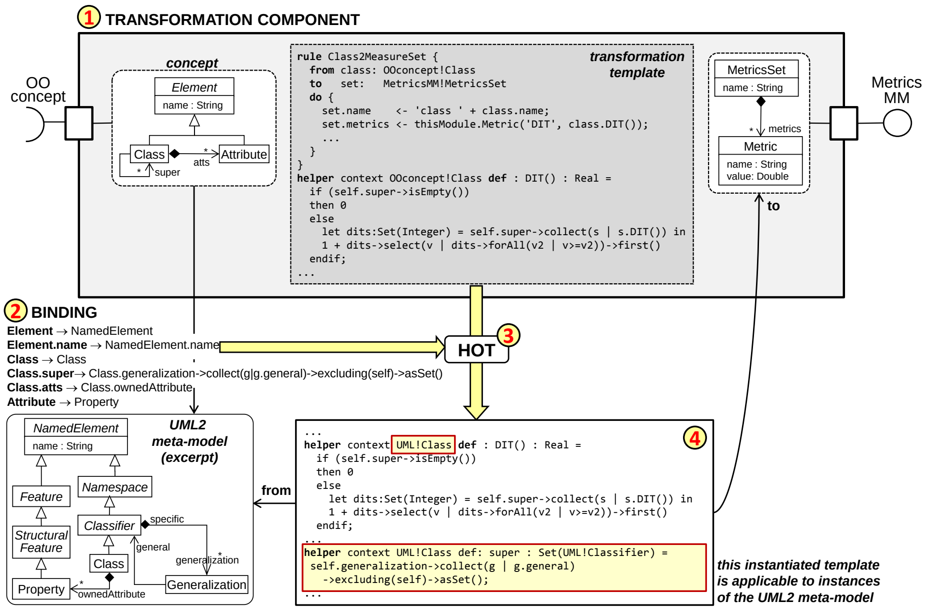

Fig. 1. Example of definition and usage of transformation components.

Tree (DIT) metric. The concept, which is the component interface, gathers the structural requirements that a meta-model needs to fulfil to qualify as input meta-model for the transformation. The concept should be as simple as possible to facilitate reuse, excluding elements that are not needed by the transformation. In the example, the concept includes class Attribute even if it is not used in the excerpt of the ATL template, because other rules do it.

The way to reuse a component is to bind its concepts to meta-models (label 2). While it is possible to have concepts as source and target of a transformation template, binding only the source is more common in practice [13]. If a concept is not bound, it is simply treated as a meta-model. By default, each element in the concept must be bound to one meta-model element. This can be adjusted for each concept element by attaching a cardinality that indicates how many times it may be bound. By space constraints, we do not discuss this feature further.

In the figure, the source concept is bound to the UML2 meta-model. The binding is performed through a dedicated domain-specific language which allows defining correspondences. The left of each correspondence refers to a concept element, like Element, Class or Class.super. The right may include either elements of the bound meta-model or OCL expressions defined over the meta-model. For example, Element is bound to NamedElement, Class to Class, and reference Class.super is bound to a collection of Class obtained through the OCL expression Class.generalization->collect(...). We use a structural approach, so that abstract classes in the concept may not need to be bound, in which case, any feature defined in it should be bound in its concrete subclasses. Thus, if Element were not bound in the example, then name should be bound in both Class and Attribute.

The binding induces an adaptation to the transformation template (label 3), yielding an ATL transformation applicable to the instances of the bound UML2 meta-model. In this case, the adaptation modifies the context of the helper, and adds a new helper that calculates the superclasses of a given one (relation super in the original concept), as given by the binding of Class.super.

Altogether, building a reusable component involves the development of a transformation template and its associated concepts from scratch. In the example, we developed a transformation to calculate metrics and a concept for object-oriented languages. However, the ATL zoo already contains two transformations that calculate object-oriented metrics for KM3 and UML2. Unfortunately, these meta-models (especially UML2) contain a lot more elements than the transformation needs, thus not being suitable to be used as concepts. If we would have been able to make reusable one of these transformations, we would have saved a lot of effort, as the resulting component would be applicable to any object-oriented modelling language. Thus, in the rest of the paper, we present a proposal to automate the reverse engineering of existing transformations into reusable components. As running example, we will reverse-engineer the transformation excerpt of Fig. 1, defined over the UML2 meta-model.

3 Making existing transformations reusable

Promoting existing transformations into reusable components poses several challenges. First, we need to simplify the used meta-models (e.g. UML2) into concepts. This process can be automated by calculating the effective meta-models of the transformation (i.e. the classes and features accessed by the transformation code). However, the effective meta-model might not be the ideal concept, as we may like e.g., to merge classes or reorganize the inheritance hierarchy. Doing this manually can be cumbersome, since it must be checked if the change breaks the transformation behaviour (i.e., adapting the transformation would imply removing a rule), and then changing the transformation accordingly if needed. In this section, we introduce our proposal to automate this process.

Fig. 2 shows the steps in our approach. First, the transformation to be made reusable is selected. This implies looking up potential sources of interesting transformations, such as in-house developed transformations, transformation repositories (e.g. the ATL zoo) and open source MDE tools that include model transformations (e.g. MoDisco and Fornax). However, not any transformation is adequate to be generalised into a generic transformation (although they still can profit from the process to improve its quality and be deployed in a repository). For example, the *Ant to Maven* transformation fully depends on the Ant and Maven semantics, and thus it cannot be generalized to other build systems. Intuitively, we say that a transformation is amenable to reuse when there are variants of the meta-models it uses (e.g. variants of UML class diagrams, different versions of it, or meta-models for related notations, like Ecore).

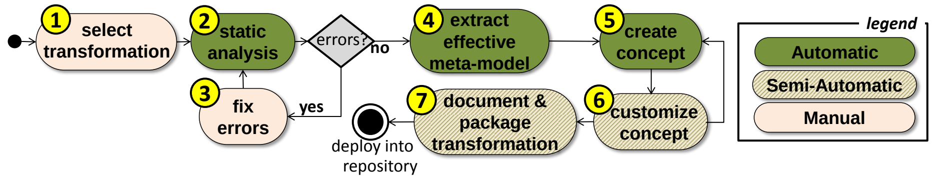

Fig. 2. Main steps in the reverse engineering of transformations into reusable units.

The second step performs a static analysis of the transformation. This is particularly needed in ATL because ATL does not enforce type correctness, hence transformations may be ill-typed. Moreover, the creation of a suitable concept for the transformation requires precise type information. If the analysis detects errors, the developer is required to fix them (step 3), otherwise, the effective meta-model of the transformation is automatically extracted (step 4).

Starting from the effective meta-model, a concept is derived, which includes the *minimum* structural requirements that a meta-model should fulfil to be used with the transformation (step 5). The concept is more concise than the effective meta-model, as it is refactored taking into account the static analysis of the transformation, e.g., to remove unused features and intermediate abstract classes, or to move features up or down class hierarchies. The aim is having a concept as simple as possible to facilitate its reuse. The suitable refactorings are automatically computed and the user is only requested to approve them. For example, if we start from a transformation defined over the UML2 meta-model, the system may suggest replacing class Generalization by a reference parents, as this will facilitate future bindings. Additionally, it is possible to customize the concept through user-selected refactorings allowing, e.g., merging two classes into one, or changing an enumerate attribute by a set of subclasses (step 6). In this way, designers can include tacit knowledge of the domain in the design of the concept. A common example is the renaming of classes to assign names more akin to the domain. Both the creation and the customization of concepts may imply the automatic rewriting of the transformation to keep it consistent, and they are iterative since the application of a refactoring may enable another one.

The final step (label 7) is to document and package the concepts and transformation template. This can be done using a variety of formats, including text- and contract-based documentation. Currently, we use the PaMoMo language to describe the transformation contract via pre/post-conditions and invariants [6], enriching the transformation with documentation and an automatically generated test suite. For space constraints, we leave out this step of the process.

4 Static analysis of ATL transformations

Our reverse engineering procedure needs to extract the *static meta-model footprint* of a transformation. This requires static type information regarding the classes and features used by the transformation. In strongly typed languages

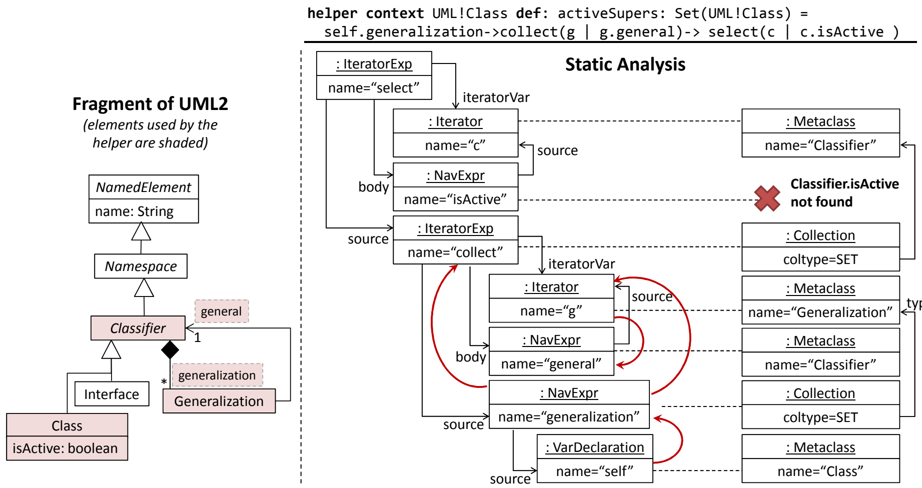

Fig. 3. Analysis of a helper.

like Kermeta [15], the type information is available in the abstract syntax tree, but other languages, like ATL, do not provide this information. Hence, as a prerequisite to apply our approach, we introduce a static analysis stage to gather as much type information as possible from the ATL transformation. This section presents the static analyser that we have built for this purpose.

In the simplest setting, our analyser performs a bottom-up traversal of the abstract syntax tree, propagating types from leaf nodes to the root of the expressions (i.e. using synthesized attributes). In some cases, particularly for parameters, types also need to be passed top-down (i.e. using inherited attributes).

Fig. 3 shows the analysis of an ATL helper which gathers the "active" direct superclasses of a class. Each node of the abstract syntax tree is annotated with its type (boxes to the right linked with dashed lines to the nodes). These types are propagated along the nodes, as depicted by the red, curved lines (we only show some of them). For nodes corresponding to a property access or operation invocation, the existence of the property or operation is checked.

Even though the helper is accepted by the ATL engine, the analyser reports a warning because the isActive property is defined in Class but not in Classifier, which is the type of the variable c. At runtime, this expression will fail if the model includes classifiers different from classes, like Interface objects.

Reporting these issues is important to help improving the quality of the transformation and understand its constraints if they are not documented. Moreover, the analyser should avoid raising false warnings and errors which may lead to low-quality type information. To this end, we have enhanced the basic analysis with the following features:

– **Multiple type collections**. In OCL, it is possible to mix objects of different, unrelated types in the same collection, typically through the use of union and including operations. Our analyser keeps track of these operations in order to: (a) infer common supertypes, or (b) assign multiple potential types to the same expression node. This provides more accurate typing information for the effective meta-model extraction and concept creation phases.

- **Implicit casting**. ATL does not support the oclAsType operation, which complicates the analysis as there is no explicit way for downcasting. Thus, our analysis looks for oclIsKind/oclIsTypeOf expressions that implicitly downcast a reference. For instance, the following variants of the expression in Fig. 3 are deemed correct by our analyser, because the usages of oclIsKindOf ensure that the type of c will be Class when used in the c.isActive expression.

```
self.generalization−>collect(g | g.general)−>
  select(c | c.oclIsKindOf(UML!Class))−>
  select(c | c.isActive )
```

```
self.generalization−>collect(g | g.general)−>
  select(c | if c.oclIsKindOf(UML!Class) then
             c.isActive else false endif)
```

- **Structural type inference**. As explained above, a property access may not be resolved due to the lack of downcasting (either explicit or implicit). In such a case, our analyser looks for the property in the subclasses of the receptor's type. If it is found in one or more subclasses, they are tentatively assigned to the expression, and a warning is raised.

This list is not exhaustive, and we aim at improving the analyser since, the better it gets, the more accurate the reverse engineering process will be. Indeed, any other ATL analyser could be used instead of ours, whenever it provides the meta-model footprint of the transformation. The meta-model footprint refers to the meta-model elements involved in the transformation. This corresponds to the set of used types, in the example {Class, Classifier, Generalization}, and the set of used features, in the example {Classifier.generalization, Generalization.general, Class.isActive} (see Fig. 3). For practical purposes, we distinguish two kinds of used types: *explicit types* if they are explicitly mentioned in the transformation, and *implicit types* if they are indirectly reached through navigation expressions.

Additionally, our analyser outputs information about *call sites*, which are the locations where an operation or feature is accessed. This is the concrete class that receives the feature access, which may be different from the class defining the feature. Thus, for each call site, we store a pair of concrete class and feature. In the example, the set of call sites is {⟨Class, Classifier.generalization⟩, ⟨Generalization, Generalization.general⟩, ⟨Class, Class.isActive⟩}. This provides more information than just the accessed features, since it is possible to know that the Classifier.generalization feature is only accessed by Class objects.

5 Creation and customization of concepts

From the information extracted in the static analysis phase, we infer a concept that will act as interface for the reusable component. For this purpose, first we prune the meta-model to keep only the elements needed by the transformation.

```
rule Class2MeasureSet {
  from class: UML!Class
  to set: MetricsMM!MetricsSet
  do {
    set.name <− 'class ' + class.name;
    set.metrics <− thisModule.Metric('DIT', class.DIT());
  }}

helper context UML!Class def : DIT() : Real =
  if (self.super−>isEmpty())
  then 0
  else
    let dits:Set(Integer) = self.super−>collect(s | s.DIT()) in
    1 + dits−>select(v | dits−>forAll(v2 | v>=v2))−>first()
  endif;

helper context UML!Class def: super : Set(Classifier) =
  self.generalization−>collect(g | g.general)
    −>excluding(self)−>asSet();
```

Meta-model Footprint

Explicit types:
- Class

Implicit types:
- Classifier
- Generalization

Features:
- NamedElement.name
- Classifier.generalization
- Generalization.general

Call sites:
- ⟨Class, NamedElement.name⟩
- ⟨Class, Classifier.generalization⟩
- ⟨Generalization, Generalization.general⟩

Fig. 4. ATL transformation over UML2 meta-model (left). Source meta-model footprint obtained after the static analysis (right).

Then, we convert the pruned meta-model into a concept which may be simplified through the application of several refactorings, and customised to take into account specific knowledge of the domain.

5.1 Extraction of effective meta-model

To calculate the effective meta-model, we use a pruning algorithm like the one presented in [15], using the meta-model footprint obtained in the static analysis as input. The algorithm keeps in the meta-model the implicit and explicit types, and respects the inheritance hierarchies.

As an example, Fig. 4 shows a transformation defined over the UML2 meta-model, and the footprint that our static analysis returns. This footprint is used to extract the effective meta-model of the transformation. In particular, Fig. 5(a) shows an excerpt of UML2, while Fig. 5(b) shows the effective meta-model that results from applying the pruning algorithm to the transformation in Fig. 4.

5.2 Concept creation

The effective meta-model is refactored into a more compact concept by removing or simplifying non-essential elements for the transformation, for which we take into account the call site information. On the one hand, the concept is the interface for reusability, and hence large inheritance hierarchies are discouraged because they affect comprehensibility [3]. On the other hand, concepts should be as simple as possible to facilitate their binding to meta-models. For example, the effective meta-model in Fig. 5(b) is not a suitable concept yet, because it contains some classes (like NamedElement) which may be not found in every object-oriented notation. This class appears in the effective meta-model because it is a container for name, which is only used by Class in the transformation.

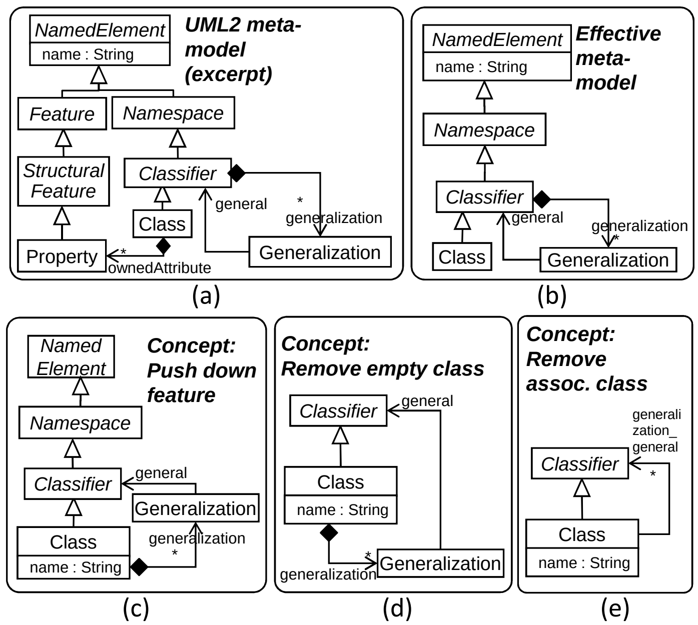

Fig. 5. Sequence of operations to convert the UML2 meta-model into a concept.

To help creating the concept, we make available a number of refactorings automating the identification of simplification opportunities, their application, and the co-evolution of the transformation whenever it is needed. The system automatically suggests refactoring opportunities to the user, along with an explanation of the rationale of the proposal and its consequences (e.g. the transformation must be co-evolved). The user only needs to approve their application, since the refactoring locations are automatically gathered. Some of the refactorings are likely to be always accepted, such as removing empty classes. Hence, our tooling allows the user to configure which refactoring opportunities should be applied automatically. Moreover, the refactorings are applied in an iterative fashion, since the application of a refactoring may yield new refactoring opportunities.

– **Push down feature**. It moves a feature defined in a class to one or more of its subclasses, if only the instances of such subclasses use the feature. This information is taken from the call sites computed in the analysis phase. The refactoring is parameterized with the maximum number of subclasses to which the feature can be moved, in order to prevent duplication of the same feature in too many subclasses. For example, according to the call sites, the NamedElement.name and the Classifier.generalization features are only used by Class instances, thus they are moved to Class (see result in Fig. 5(c)).
– **Remove empty class**. Classes without features are removed if they do not belong to the *explicit types* set (i.e. they are only used in navigation expres-

sions). If the removed class is both a subtype and a supertype, the inheritance relationships are rearranged (this is called *pull-up inheritance* in [4]). The goal of this refactoring is to collapse inheritance hierarchies to enhance the comprehensibility of the concept and facilitate future bindings. In the running example, Namespace is removed, as well as NamedElement because the previous refactoring "pushed down" its only feature (see Fig. 5(d)).

- **Remove unused feature**. Any feature appearing in the effective meta-model but not in the footprint is removed. This is needed because the pruning algorithm [15] leaves opposite references even when they do not appear in the effective meta-model. Thus, this refactoring refines the pruning algorithm.
- **Make leaf abstract class concrete**. The effective meta-model may include leaf abstract classes, if their subclasses do not belong to the set of explicit types. In such a case, this refactoring makes such classes concrete, thus enforcing their binding to some class in the bound meta-models.
- **Pull up feature**. If several subclasses with a common parent share features, these are pulled up to the parent. This situation can arise initially in the effective meta-model, or due to the application of other refactorings. The refactoring can be parameterized with the minimum number of classes that should define the feature in order to pull it up.
- **Remove association class**. An association class acts as a reference that is able to carry properties. A typical example is Generalization in the UML2 meta-model. If a transformation does not use the properties of an association class (except the reference to the target class, like general in UML2), and the class does not appear in the *explicit types* set (except when used in *allInstances* operation), then the association class can be replaced by a simple reference in the concept. In such a case, the transformation needs to be co-evolved, replacing the navigations through the association class by references. The benefit of this refactoring is two-fold. Firstly, the concept becomes simpler. Secondly, the binding will be simpler if the meta-model also represents the same element as a reference, whereas if not, binding a reference in a concept to a class in a meta-model is easier than the other way round (we just need an expression like the one in Fig. 1 for Class.super). Fig. 5(e) shows its application to the concept, which implies co-evolving the transformation template. The details of the transformation rewriting are left out due to space constraints. In the running example, the expression self.generalization->collect(g | g.general)->excluding(self)->asSet() gets rewritten into self.generalization_general->excluding(self)->asSet().

5.3 Concept customization

The previous process yields a concept, simplified to make it concise and reusable. However, this concept still retains the nomenclature and some design decisions from the meta-model from which it was derived. At this point, domain expertise can be used to customise the concept so that it reflects tacit knowledge of the domain. A typical example is the renaming of classes and features using the terms most frequently used in the domain. Similarly, some design options may be more common in a particular domain than others.

Next, we enumerate the domain-specific customizations currently supported, some of them inspired by standard object-oriented refactorings [5]. Some refactorings induce an adaptation of the transformation template, or use the information extracted from the static analysis of the transformation:

- **Renaming of classes and features**. It changes the name of classes and features, rewriting the transformation to accommodate the new names.
- **Extract sub/superclass**. This is a pair of related refactorings. *Extract subclass* splits a class into a superclass/subclass pair, the former optionally abstract. *Extract superclass* creates a new abstract superclass for a given set of classes, pulling up their common features. In both cases, the transformation does not need to be adapted.
- **Collapse hierarchy.** This refactoring merges a class and a child class. It can only be applied if the parent class is not an *explicit type*, and it has just one child. This refactoring does not rewrite the transformation. However, if the concept includes some reference to the superclass, then the user is warned that if the superclass is bounded to a meta-model class with several children, collapsing the hierarchy excludes those children from the reference. For example, this refactoring is applicable in Fig. 5(e) because Classifer is not an explicit type and has a unique child Class. The result is a concept with a single node Class and a self-reference generalization_general. In this case, a warning is issued because Classifier received a reference. This means that if the resulting concept is bound back to the UML2 meta-model, mapping Class in the concept to Class in the meta-model, the reference generalization_general will only contain Class objects. Instead, if we keep the concept in Fig. 5(e) and map both Classifiers in the concept and the meta-model, then the reference may hold any subclass of Classifier (Class objects but also Interface objects).
- **Replace enumerate with inheritance**. An enumeration attribute used to distinguish several class types is replaced by a set of subclasses, one for each possible value. This refactoring is applicable if the enumeration literals are only present in comparisons, getting substituted by oclIsKindOf(...).

This list is not exhaustive, as we are working on additional ones, taken from [5]. As a difference from the refactorings presented in the previous section, the identification of the customization opportunities is not automated as it is difficult to deduce, e.g., whether the name of a class is appropriate in a domain or if a certain notion is better represented using two classes instead of one. Thus, users must select the locations where a customization should be performed, and then the concept is changed accordingly and the transformation is automatically adapted when possible.

6 Evaluation and tool support

We have evaluated our approach along two dimensions, described by the following two questions. First, *can we obtain a reusable component from a transforma-*

	Process	DSC	NOH	ANA	ADI	NAC
UML	Initial meta-model	247	246	6.91	5.60	48
UML	Compute effective meta-model	31	30	0.77	0.47	23
UML	Ref. remove empty class (14)	17	15	0.59	0.36	9
UML	Ref. make abstract class concrete (1)	17	15	0.59	0.36	8
UML	Ref. push down feature (5)	17	15	0.59	0.36	8
UML	Ref. remove empty class (2)	15	13	0.53	0.27	6
UML	Ref. remove association class (1)	14	13	0.57	0.29	6
KM3	Initial meta-model	16	16	0.31	0.07	2
KM3	Compute effective meta-model	11	10	0.45	0.01	2
KM3	Ref. remove empty class (1)	10	9	0.4	0.01	1
KM3	Ref. push down feature (1)	10	9	0.4	0.01	1

DSC: design size in classes
NOH: number of hierarchies
ANA: average number of ancestors
ADI: average depth of inheritance
NAC: number of abstract classes

Fig. 6. Metrics taken at each step of the process, for UML2 and KM3. The number of applications of each refactoring is shown between parentheses.

tion not designed to be reused?. Second, *to what extent is the effective meta-model simpler than the original one, and the concept simpler than the effective meta-model?*. To answer these questions, we have made an experiment based on two transformations from the ATL zoo, which calculate object-oriented metrics, one for UML2 (UML2Measure) and the other one for KM3 (KM32Measure).

To answer the first question, we applied our reverse engineering process to UML2Measure. We obtained a concept which we were able to bind to other object-oriented notations like KM3, Ecore, Java/Jamopp and MetaDepth. The bindings have less than 40 LOC, whereas the original transformation has about 370 LOC. This shows that our technique is effective, and yields reusable transformation components with concise concepts as interface for reuse.

To answer the second question, we reverse engineered both transformations and measured the effective meta-models/concepts obtained along the process. We used the object-oriented metrics proposed in [3], related to understandability and functionality quality attributes. High values of these metrics influence negatively the understandability. Fig. 6 summarizes the results. For UML2Measure, computing the effective meta-model removes all classes not related to class diagrams; however, the metrics relative to hierarchies and abstract classes indicate that the effective meta-model still has complex hierarchies. Our refactorings reduce this complexity to the half, obtaining a concept significantly simpler than the meta-model. In the case of KM32Measure, the computation of the effective meta-model and the refactorings have less impact because KM3 is a very simple meta-modelling core, almost a concept.

We also evaluated the gain from using the final, refactored concept as interface for reuse, w.r.t. using the effective meta-model for that purpose. Thus, we reused UML2Measure for Ecore, KM3, Java/Jamopp and MetaDepth. In all cases, the bindings from the concept were simpler than from the effective meta-model. For instance, abstract classes can be left unbound in our approach; but since the effective meta-model contained lots of them, the burden to decide what to bind to what was much lower for the concept. The *push down feature* refactorings improved the comprehensibility of the concept, because features were no longer hidden in the middle of hierarchies. The *remove association class* refactoring was particularly useful, as none of the bound meta-models had the notion of Generalization present in UML2. Thus, we had to define fairly complex bindings from the effective meta-model to emulate the Generalization class, but the bindings from the concept were straightforward. Altogether, this experiment shows that the obtained concept favours reuse more than the effective meta-model. A more extensive evaluation to confirm this intuition is left for future work.

Additionally, we validated the correctness of our implementation, binding the concept obtained from the UML2Measure transformation to the original UML2 meta-model. Then, we executed the original transformation and the adapted template using several third-party UML models as input, checking with EMF Compare that the results were in fact the same.

To support our reverse engineering process, we have built an Eclipse plug-in integrated in the Bentō[1] tool. The tool is interactive. As an example, Fig. 7 shows part of the interaction for the KM32Measure case: (1) the original transformation and the component information is configured, (2) the analysis phase detects warnings and errors in the transformation, (3) the refactoring opportunities are listed and can be easily applied, displaying the result in a tree-based visualization. Step (3) can be repeated if the system finds new refactoring proposals due to the application of a previous refactoring, or to apply domain customizations. To support this step, the tool allows computing metrics and showing information about the use of the concept in the transformation (4). Interestingly, the metrics facility has been included by reusing the oo2measure component obtained in the evaluation, and binding it to Ecore. Finally, the component is packaged by generating meta-information for our Bentō tool (5).

f

7 Related work

Proposals on model transformation reuse can be type-centric or type-independent. The former include reuse mechanisms for single rules, like rule inheritance [11], and for whole transformations, like superimposition [19] and phases [14]. Regarding type-independent approaches, there are fine-grained techniques like parameterized rules [8, 10, 17], and coarse-grained ones aimed at reusing complete transformations [16]. Among these proposals, only [16] supports the reuse of transformations for arbitrary meta-models, as in our case. For this purpose, the authors extract the effective meta-model of the transformation as-is, and adapt the meta-model where the transformation is to be reused by making it a subtype of the effective meta-model. In contrast, we use concepts as reuse interface, we simplify the effective meta-model to facilitate its binding, and we do not modify the models/meta-models to be transformed but we adapt the transformation.

Our approach performs a static analysis of the original transformation. Even though the ATL IDE includes a static analysis engine that proposes feature completions, this only provides basic information which is not very accurate. The static analyser presented in [18] allows navigating ATL transformation models.

[1] The tool and a screencast are available at `http://www.miso.es/tools/bento.html`.

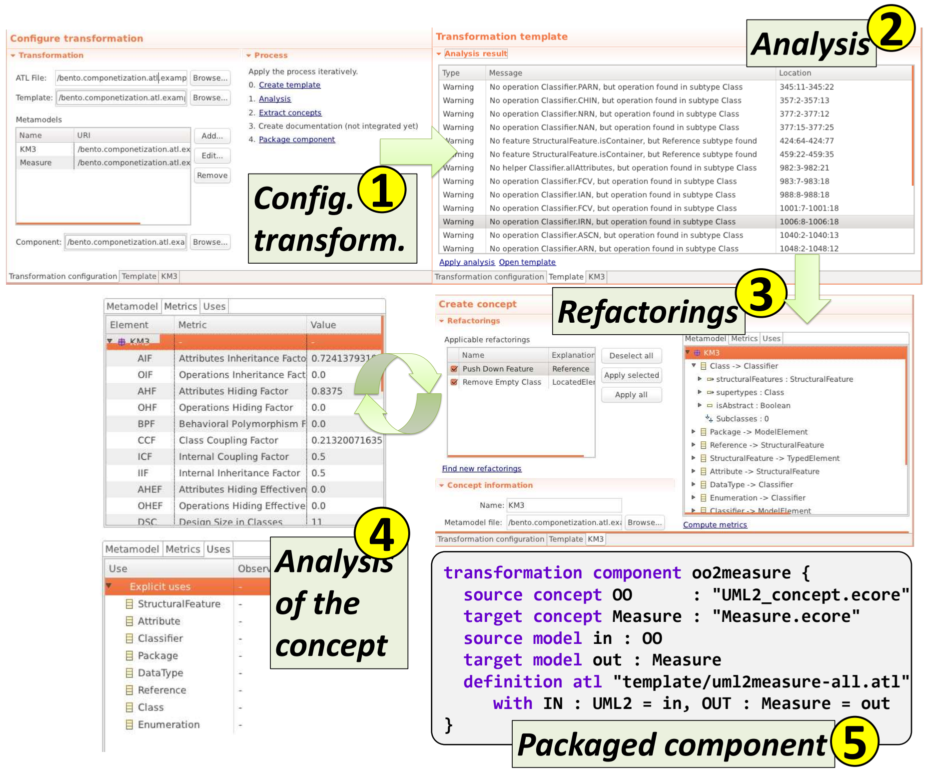

Fig. 7. Process followed to reverse engineer the KM32Measure transformation.

The analyser, which is a facade to the ATL meta-model provided as a Java API, does not provide type information or advanced analysis support.

Our meta-model extraction procedure relates to works on meta-model slicing and shrinking, though our goal is to simplify a meta-model to make an associated transformation easier to reuse. This poses additional challenges, like the need to identify whether a meta-model refactoring does not break the transformation.

Meta-model pruning is usually structure-preserving. For instance, the algorithm presented in [15] takes a set of elements of interest of a meta-model (in our case the meta-model footprint of the transformation) and returns a pruned version of the meta-model containing the minimum set of elements required for the new version to be a subtype of the original. Our approach is similar, but we simplify the resulting meta-model, e.g., by flattening hierarchies and removing opposite features unless both ends belong to the meta-model footprint. In [7], static meta-model footprints are obtained from Kermeta code in order to estimate model footprints. Kermeta includes type information in the syntax tree, hence no explicit static analysis is needed. The meta-model pruning phase is in line with [15], except that it includes all subclasses of every selected class.

A few works propose simplification techniques for meta-models, mostly based on refactorings for object oriented systems [5]. For instance, in [4], the authors present some type-safe meta-model reduction operations which guarantee extensional equivalence between the original and the reduced meta-model (i.e. the set of models conforming to both meta-models is the same). Their approach computes the meta-model snippet needed to represent a selection of classifiers and features from a set of initial models, and then applies several type-safe reduction operations to the meta-model snippet. As reduction operations, they support the flattening of hierarchies and the removal of features declared by classifiers which were not explicit in the initial models. Type-safety is achieved through the *pull-up inheritance*, *push-down feature* and *specialize feature type* refactorings [5]. In our case, we obtain the meta-model footprint through the static analysis of the transformation, which is more challenging. While we support the same reduction operations (among others), their applicability is restricted by the transformation, which may prevent some changes. Moreover, we provide further refactorings whose goal is to facilitate the binding of the concept, and may induce the transformation adaptation.

Some of our concept refactorings require adapting the transformation, like in meta-model/transformation co-evolution [9, 12]. These works distinguish three kinds of transformation changes: fully automated, partially automated and fully semantic. In our case, we only consider meta-model changes that lead to fully automated transformation changes, as we aim at an automated process. In contrast to [9, 12], we use typing information derived from the transformation.

Altogether, to the best of our knowledge, this work is the first attempt to reverse-engineering model transformations for enhancing their reusability.

8 Conclusions

In this paper, we have presented our approach to reverse engineer existing transformations into reusable components that can be applied to different meta-models. For this purpose, we first perform a static analysis of the candidate transformation to extract typing information and identify type errors. Then, we use this information to build a concept, that is, an interface optimised and customised to facilitate the reuse of the transformation. In this process, the transformation may need to be adapted to make it conformant to the concept.

We have demonstrated our approach and supporting tool by performing the reverse engineering of an existing ATL transformation to calculate object-oriented metrics. The results show that the obtained concepts tend to be more concise than meta-models, and therefore suitable for our purposes.

In the future, we foresee having a repository of reusable components that can be navigated and integrated with other components, thus speeding up the development of MDE projects. In addition to support reusability of whole transformations, we will also consider extracting slices of an existing transformation, and its subsequent re-engineering into a reusable component. We would like to consider other kinds of components, like components for code generation or inplace transformation, as well as further transformation languages in addition to ATL. While we support the manual definition of PaMoMo specifications for documenting transformation components, we plan to work on their automatic derivation from existing transformations. Such specifications could be used as composability criteria for components and for testing.

Acknowledgements. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139).

References

1. A. Kusel et al. Reuse in model-to-model transformation languages: are we there yet? *SoSyM*, pages 1–36, 2013.
2. AtlanMod. Atl zoo. `http://www.eclipse.org/atl/atlTransformations/`.
3. J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Software Eng.*, 28(1):4–17, 2002.
4. A. Bergmayr, M. Wimmer, W. Retschitzegger, and U. Zdun. Taking the pick out of the bunch - type-safe shrinking of metamodels. In *SE'13*, pages 85 – 98, 2013.
5. M. Fowler. *Refactoring. Improving the Design of Existing Code.* Ad.-Wesley, 1999.
6. E. Guerra and M. Soeken. Specification-driven model transformation testing. *SoSyM*, pages 1–22, 2013.
7. C. Jeanneret, M. Glinz, and B. Baudry. Estimating footprints of model operations. In *ICSE'11*, pages 601–610. ACM, 2011.
8. E. Kalnina, A. Kalnins, E. Celms, and A. Sostaks. Graphical template language for transformation synthesis. In *SLE'09*, volume 5969 of *LNCS*, pages 244–253. Springer, 2010.
9. T. Levendovszky, D. Balasubramanian, A. Narayanan, and G. Karsai. A novel approach to semi-automated evolution of dsml model transformation. In *SLE*, volume 5969 of *LNCS*, pages 23–41. Springer, 2009.
10. M. Wimmer et al. Surviving the heterogeneity jungle with composite mapping operators. In *ICMT'10*, pages 260–275, LNCS 6142, 2010. Springer.
11. M. Wimmer et al. Surveying rule inheritance in model-to-model transformation languages. *JOT*, 11(2):3:1–46, 2012.
12. D. D. Ruscio, L. Iovino, and A. Pierantonio. A methodological approach for the coupled evolution of metamodels and atl transformations. In *ICMT*, volume 7909 of *LNCS*, pages 60–75. Springer, 2013.
13. J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Flexible model-to-model transformation templates: An application to ATL. *JOT*, 11(2):4:1–28, 2012.
14. J. Sánchez Cuadrado and J. G. Molina. Modularization of model transformations through a phasing mechanism. *SoSyM*, 8(3):325–345, 2009.
15. S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model pruning. In *MoDELS'09*, volume 5795 of *LNCS*, pages 32–46. Springer, 2009.
16. S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. *SoSyM*, 11(1):111–125, 2010.
17. D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *UML'04*, volume 3273 of *LNCS*, pages 290–304. Springer, 2004.
18. A. Vieira and F. Ramalho. A static analyzer for model transformations. In *3rd International Workshop on Model Transformation with ATL*, 2011.

19. D. Wagelaar, R. V. D. Straeten, and D. Deridder. Module superimposition: a composition technique for rule-based model transformation languages. *SoSyM*, 9(3):285–309, 2010.