Softw Syst Model
DOI 10.1007/s10270-013-0343-7

THEME SECTION PAPER

# Reuse in model-to-model transformation languages: are we there yet?

**A. Kusel · J. Schönböck · M. Wimmer · G. Kappel · W. Retschitzegger · W. Schwinger**

Received: 31 October 2012 / Revised: 27 March 2013 / Accepted: 2 April 2013
© Springer-Verlag Berlin Heidelberg 2013

**Abstract** In the area of model-driven engineering, model transformations are proposed as the technique to systematically manipulate models. For increasing development productivity as well as quality of model transformations, reuse mechanisms are indispensable. Although numerous mechanisms have been proposed, no systematic comparison exists, making it unclear, which reuse mechanisms may be best employed in a certain situation. Thus, this paper provides an in-depth comparison of reuse mechanisms in model-to-model transformation languages and categorizes them along their intended scope of application. Finally, current barriers and facilitators to model transformation reuse are discussed.

**Keywords** Reuse mechanisms · Model transformations · Survey · Model-driven engineering

Communicated by Prof. Zhenjiang Hu.

A. Kusel · W. Retschitzegger · W. Schwinger
Johannes Kepler University Linz, Linz, Austria
e-mail: Angelika.Kusel@jku.at

W. Retschitzegger
e-mail: Werner.Retschitzegger@jku.at

W. Schwinger
e-mail: Wieland.Schwinger@jku.at

J. Schönböck (✉)
Upper Austrian University of Applied Sciences, Hagenberg, Austria
e-mail: johannes.schoenboeck@fh-hagenberg.at

M. Wimmer
University of Málaga, Málaga, Spain
e-mail: mw@lcc.uma.es; Wimmer@big.tuwien.ac.at

M. Wimmer · G. Kappel
Vienna University of Technology, Vienna, Austria
e-mail: Kappel@big.tuwien.ac.at

## 1 Introduction

Model-driven engineering (MDE) [59] proposes an active use of models to conduct the different phases of software development. Hence, models become first-class artifacts throughout the software lifecycle, leading to a shift from the "everything is an object" paradigm to the "everything is a model" paradigm [8]. Thereby, model transformations [62] are proposed to systematically manipulate models. Thus, model transformations are crucial for the success of MDE, being comparable in role and importance to compilers for high-level programming languages, since there is a recurring need (i) to transform models between different abstraction levels to bridge the gap between design and implementation, (ii) to migrate models between different language versions, or (iii) to translate models into semantic domains [4,27]. These kinds of transformations are called model-to-model (M2M) transformations and are focused in this paper.

Given the prominent role of M2M transformations in MDE and their use in increasingly complex scenarios, appropriate reuse mechanisms are indispensable to increase development productivity as well as quality of model transformations. Although this need has been recognized by the research community, as a plethora of proposed reuse mechanisms reveals, most of today's transformation designers still follow an *ad hoc manner* to specify model transformations [32]. This is not least due to the fact that (i) the application scenarios of the different reuse mechanisms are not clearly defined, (ii) it is still unclear which M2M transformation languages support which reuse mechanisms, since diverse mechanisms have been proposed by researches, which base on different transformation languages, and (iii) to which extent, syntactically similar reuse mechanisms of different M2M transformation languages vary semantically.

Published online: 11 May 2013

To alleviate this situation, this paper provides an in-depth survey of proposed reuse mechanisms in M2M transformation languages. In this survey, first typical reuse scenarios are identified by systematically combining the dimensions of (i) the *granularity* of the reusable artifact, (ii) the *specificity* of the reusable artifact, and (iii) the *scope*, where the reusable artifact is going to be integrated. Second, to explicate the main differences as well as commonalities between the reuse mechanisms along these dimensions, a comparison framework analogous to the main phases of software reuse [44] is derived, comprising the four phases of *abstraction*, *selection*, *specialization*, and *integration*. Third, representative M2M transformation languages and reuse mechanisms are selected to discuss syntactical as well as semantical differences for providing the reader a sound basis for selecting a certain reuse mechanism and/or language. Finally, barriers and facilitators for applying current reuse mechanisms for model transformations are discussed to provide a roadmap for model transformation reuse research.

**Outline** Section 2 presents a three-dimensional space for classifying reuse mechanisms and explicating typical reuse scenarios, acting as basis for deriving a common comparison framework. This framework is used in Sect. 3 to compare reuse mechanisms and supporting languages that have been found in the literature. Based on the results of the comparison, Sect. 4 presents barriers and facilitators to reuse in M2M transformations, and finally, Sect. 5 concludes the paper.

This paper goes far beyond the work described in [82], adding the following two main contributions. First, the different reuse scenarios have been introduced by meansof a systematic categorization providing a sound basis for the proposed comparison framework. Second, in the original version, we compared the different reuse mechanisms without investigating dedicated M2M transformation languages. In contrast, in this paper, we elicit in detail the provided means for reuse in diverse M2M transformation languages.

## 2 Classifying reuse mechanisms

When investigating literature on M2M transformation reuse, a plethora of approaches may be found. However, there is a lack of comparing different approaches. Such a comparison would be highly beneficial to reason about commonalities and differences between the different approaches and may act as basis for selecting a certain approach for a specific situation. For classifying reuse mechanisms and explicating typical reuse scenarios, we introduce a *three-dimensional space* in the following.

### 2.1 Dimensions of transformation reuse mechanisms

Central to any transformation reuse mechanism is that transformation logic should be reused, which is denoted as *reusable artifact* in the following. For identifying typical reuse scenarios, the key dimensions of the reusable artifact are illustrated in Fig. 1. In this respect, first, the reusable artifact may exhibit different *granularities*, i.e., if *small* parts (rules or functions) or *large* parts (including the whole transformations themselves) are going to be reused. Furthermore, the reusable artifact may be bound to *concrete* metamodel types or not by using *generic* types, leading to the second dimension named *specificity*. Finally, the reusable artifact may be employed in different *scopes*, i.e., whether the reusable artifact is reused within a single transformation only (source and target metamodels are the same or subtypes—further referred as *intra*) or across transformation boundaries (at least one source or target metamodel is not the same or a subtype—further referred as *inter*). Please note that although at first sight there may be a strong interrelationship between the two dimensions “scope” and “granularity,” these two dimensions are nevertheless orthogonal, since, e.g., a function, which would be classified as small in granularity may be reused within a single transformation (cf. intra) or between different transformations (cf. inter).

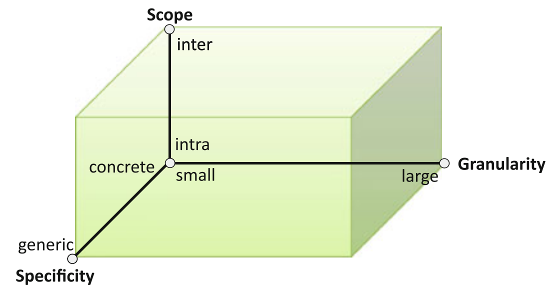


**Fig. 1** Dimensions of transformation reuse mechanisms

### 2.2 Reuse scenarios

By systematically combining the potential values of the three dimensions, eight scenarios may be identified as summarized in Table 1 and detailed in the following. For exemplifying the different reuse scenarios, additionally an example per scenario on basis of the well-known `Class2ER` example [9] is given.

#### 2.2.1 *Reuse scenario 1: concrete intra-transformation reuse in the small*

The first reuse scenario is characterized by *(intra, concrete, small)*, which targets reuse within a single transformation, whereby the reusable artifact is bound to concrete metamodel types, and the reused transformation artifact is small.

*Example* An example for this reuse scenario is depicted in Fig. 2, where a single transformation between the `Class`

**Table 1** Reuse scenarios and their reuse mechanisms

| Reuse scenario | Scope Inter/intra | Specificity Concrete/generic | Granularity Small/large | Classified reuse mechanisms |
|---|---|---|---|---|
| Scenario 1 | Intra | Concrete | small | Code scavenging, user-defined functions, rule inheritance |
| | Inter | Concrete | Small | |
| Scenario 2 | Intra | Concrete | Large | Module import, transformation product lines |
| Scenario 3 | Inter | Generic | Small | HOT, AOP, reflection, generic functions, embedded DSLs |
| | Intra | Generic | Small | |
| Scenario 4 | Inter | Generic | Large | Generic transformations, stand-alone DSLs |
| | Intra | Generic | Large | |
| Scenario 5 | Inter | Concrete | Large | Orchestration |

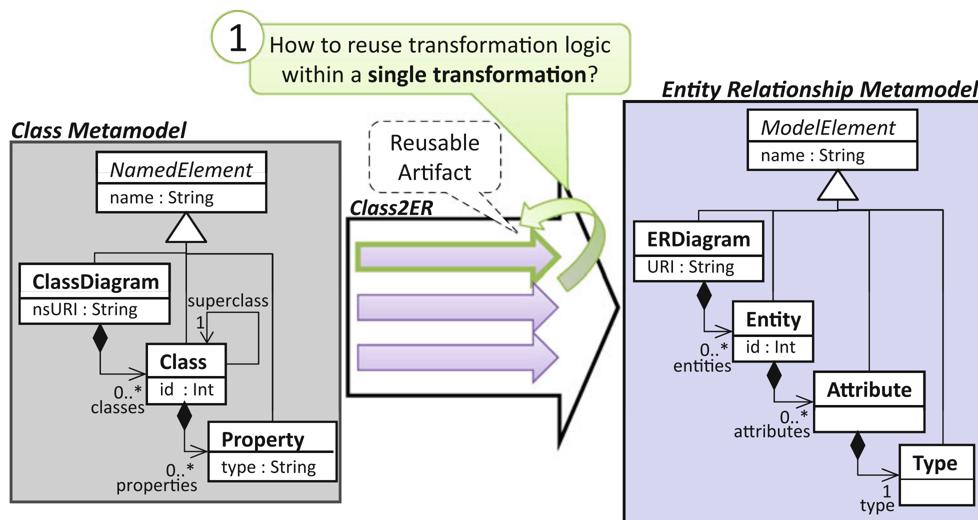


**Fig. 2** Running example—reuse scenario 1

metamodel and the `Entity Relationship` metamodel should be developed, and recurring parts of the transformation logic should be specified only once and reused accordingly. Typical reuse mechanisms which are suitable for such a scenario are *code scavenging*, *user-defined functions*, and *rule inheritance*.

### 2.2.2 *Reuse scenario 2: concrete intra-transformation reuse in the large*

The second reuse scenario is described by the attributes *(intra, concrete, large)*, which means that reuse should again take place within a transformation, whereby "*within*" means a family of similar transformations between the same metamodels. The reused artifact is bound to concrete metamodel types, and the portion of reuse is rather large.

*Example* To exemplify such a reuse scenario, Fig. 3 provides an example. In this example, a transformation `OneTablePerHierarchy` between the metamodels `Class` and `Entity Relationship` should be developed on basis of an existing transformation `OneTablePer Class` between the same metamodels. The difference between them is that different OR-mapping approaches should be realized. Reuse mechanisms facilitating such a scenario are *module import* and *transformation product lines*.

### 2.2.3 *Reuse scenario 3: generic inter-transformation reuse in the small*

The third reuse scenario exhibits the values of *(inter, generic, small)*, demanding for reuse across transformation boundaries. Furthermore, the reusable artifacts are independent of concrete metamodel types and are small.

*Example* A typical example falling into this category is shown in Fig. 4, where transformation logic independent of the concrete metamodels or transformations should be reused. An example for this is cross-cutting concerns like debugging or tracing. Reuse mechanisms that allow to realize

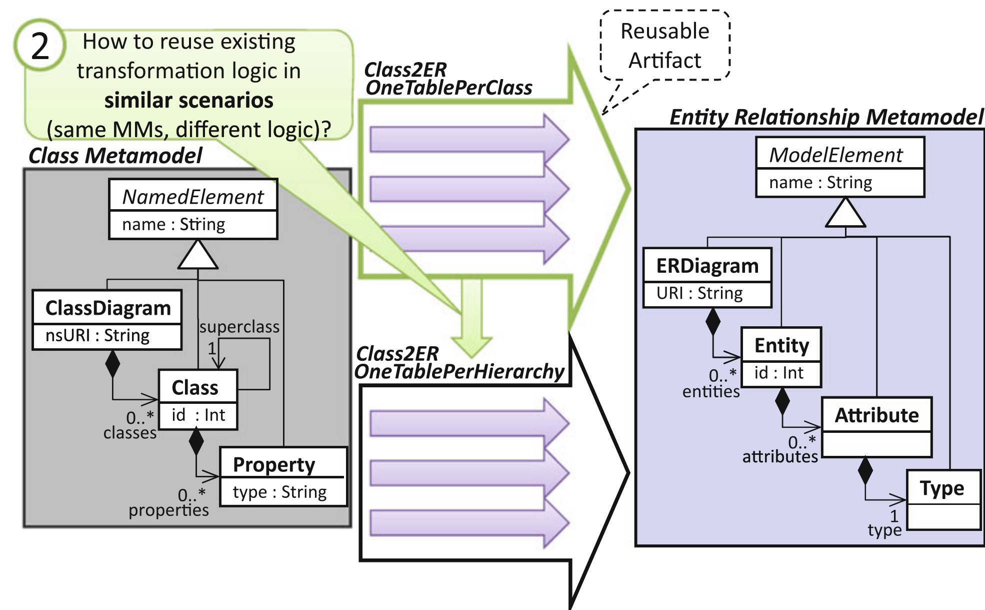


**Fig. 3** Running example—reuse scenario 2

such scenarios are *higher-order transformations* (HOTs), *aspect-orientation*, *reflection*, *generic functions*, and *embedded DSLs*.

### 2.2.4 Reuse scenario 4: generic inter-transformation reuse in the large

The fourth reuse scenario refers to the values of *(inter, generic, large)*, meaning that reuse should take place between transformations. Additionally, the reusable artifact abstracts from concrete metamodels types and is large.

*Example* To exemplify such a reuse scenario, Fig. 5 shows a simple transformation scenario, where the transformation logic of the `Class2ER` transformation should be reused in an `Ontology2XML` transformation, since the included metamodels are structurally similar. *Generic Transformations* and *stand-alone DSLs* are reuse mechanisms, which are suitable for these requirements.

### 2.2.5 Reuse scenario 5: concrete inter-transformation reuse in the large

Finally, the fifth reuse scenario includes the attributes of *(inter, concrete, large)*; thus, reuse occurs across transformation boundaries, whereby the reusable artifacts are bound to concrete metamodel types and are large.

*Example* To exemplify this reuse scenario, Fig. 6 depicts the case that a `Class2Relational` transformation is realized by reusing a `Class2ER` transformation and an `ER2Relational` transformation. A reuse mechanism which allows to realize such a scenario is the *orchestration* of existing model transformations into a more coarse-grained transformation.

### 2.2.6 Remaining combinations

Please note that three of the potential eight combinations are not explicitly discussed in this paper, because the reuse techniques for other scenarios are sufficient to be applied for them. These three comprise the combinations of (i) *(intra, generic, small)*, which would mean that, e.g., generic functions are specified for a single transformation only. We consider this combination to be solvable with techniques discussed for scenario 3. (ii) *(intra, generic, large)*, which would mean that, e.g., a whole generic transformation is specified to be reused for a set of subtype languages. Again, this scenario does not essentially require different techniques used in scenario 4. (iii) *(inter, concrete, small)*, which would mean that, e.g., concretely typed functions are reused across transformation boundaries, which would only be possible, if the transformations share at least some metamodel types. Thus, solutions for scenario 1 can be successfully applied.

This section now introduces the general process involved in reuse, which is used to derive a common comparison framework for reuse in model transformations.

## 2.3 Reuse process and conceptual comparison framework

Software reuse has been described as the "process of creating software systems from existing software rather than building software systems from scratch" [44]. In M2M transformations, reuse mechanisms focus on different application

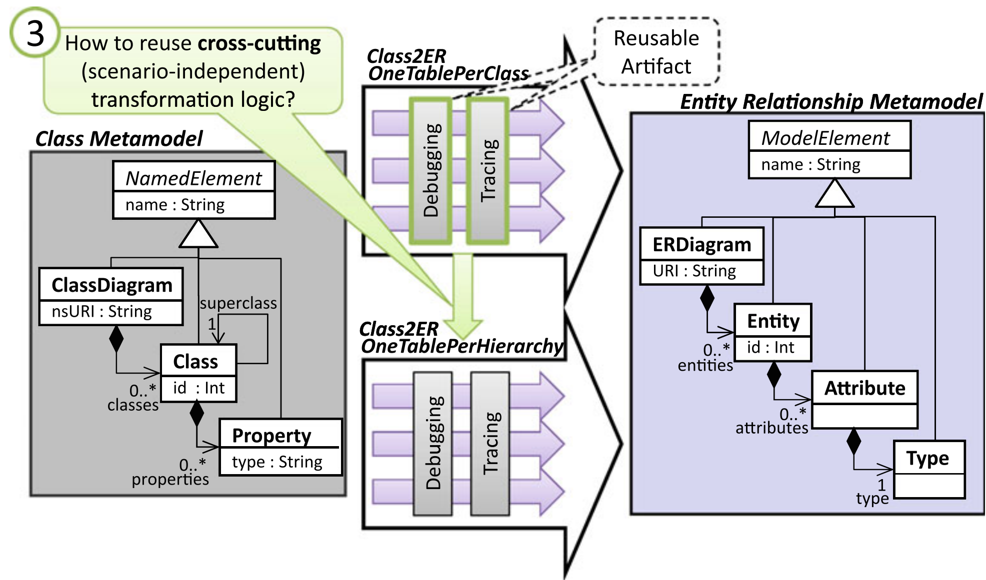


**Fig. 4** Running example—reuse scenario 3

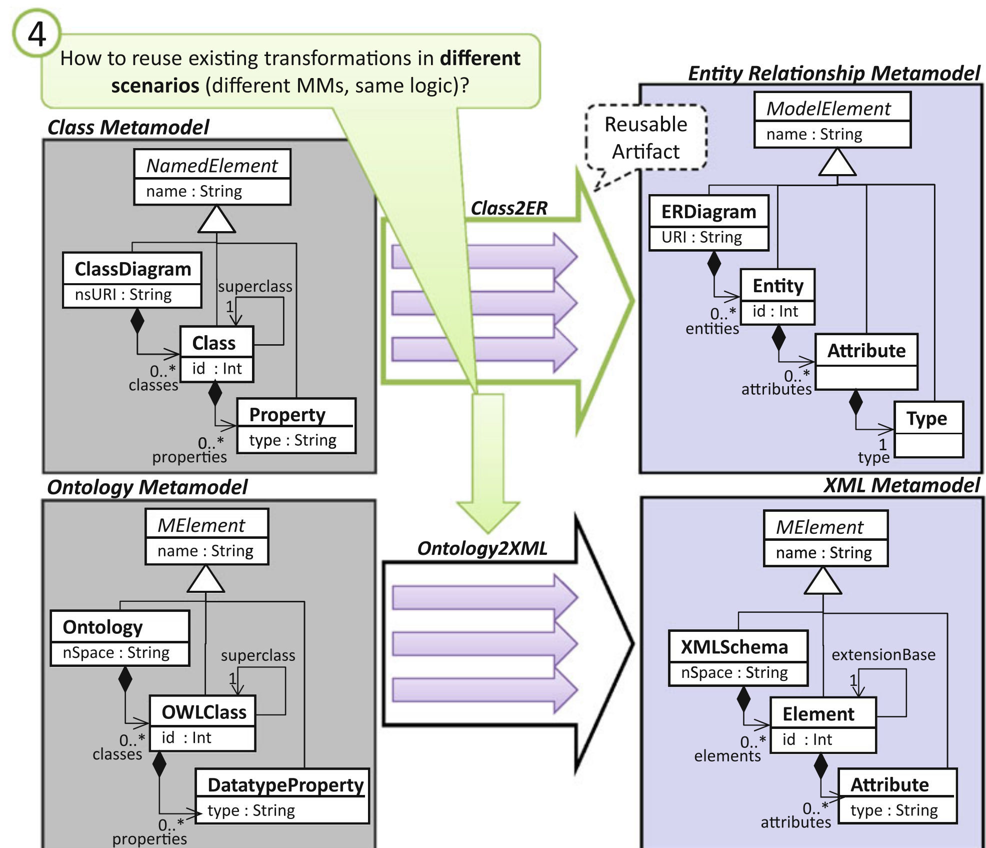


**Fig. 5** Running example—reuse scenario 4

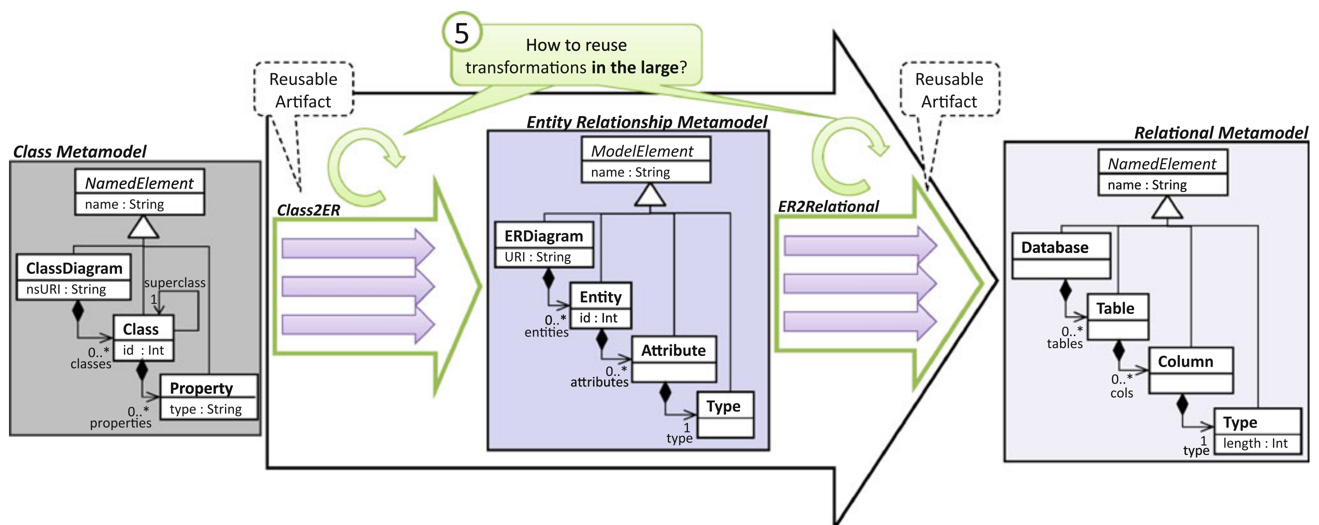


**Fig. 6** Running example—reuse scenario 5

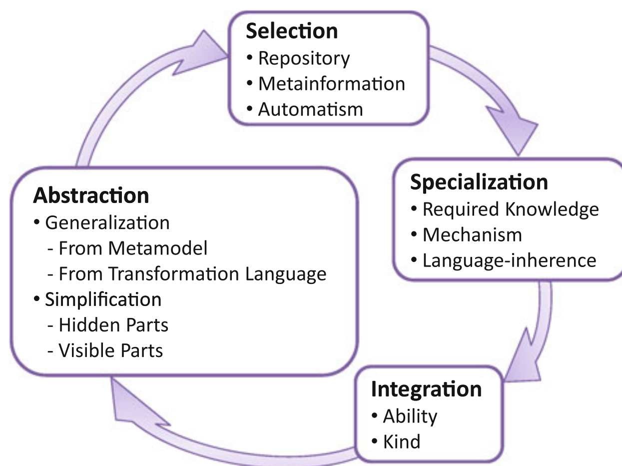


**Fig. 7** Comparison framework with comparison criteria

scenarios and thus exhibit different characteristics. However, according to [44], they follow a common process which is cyclic in nature to accommodate for continuous reuse (cf. Fig. 7). The process comprises the common phases (i) *abstraction*, (ii) *selection*, (iii) *specialization*, and (iv) *integration* and is used to derive a comparison framework (cf. Fig. 7) including dedicated criteria to characterize different reuse mechanisms, as discussed in the following.

**Abstraction** To enable reuse, abstraction is the key of any reuse mechanism. According to [43], one might distinguish between *abstraction by generalization* and *abstraction by simplification*. Abstraction by generalization allows to make an artifact reusable in different situations. To achieve this in the context of model transformations, it should be possible to decouple transformation logic from *type information*, i.e., the source and the target *metamodels*. Furthermore, reuse of transformation logic across platforms should be possible by generalizing from a certain *transformation language*. Abstraction by simplification allows to explicitly represent the information necessary for reuse, i.e., the *visible part* (e.g., the interface of a function to be reused), but to hide the actual realization of the artifact, i.e., the *hidden part* (e.g., the implementation of the function) [44].

**Selection** Provided that *repositories* of reusable artifacts exist, mechanisms are needed to efficiently find the artifacts therein. Such mechanisms range from *metainformation*, e.g., in terms of a documentation or pre-/post-conditions, to *automatism* in the form of wizards or more advanced techniques stemming from the area of information retrieval [52].

**Specialization** Specialization means the adaptation of an abstracted artifact to a specific transformation. Ideally, only *knowledge* of the interface of the abstracted artifact is needed, i.e., *black-box* reuse. In contrast, *white-box* reuse demands additional knowledge of the realization. Typical specialization *mechanisms* are, e.g., passing of parameter values to functions or overriding/extending parts in the context of inheritance. Finally, *language inherent* means may be applied for specialization, i.e., the same formalism for specialization is used as for the definition of the reusable artifact, or not.

**Integration** Whereas specialization solely configures an artifact, integration focuses on how reusable artifacts interact with the already existing parts of the specified transformation.

Integration mechanisms in software engineering are typically categorized into *composition* and *generation* mechanisms [10,53]. Thereby, composition implies that integration must take place. In contrast, generation implies that an executable transformation without further need for integration is produced. Therefore, the first criterion *ability* distinguishes between composition and generation, whereas the second criterion *kind* differentiates potential ways of composition. In this respect, according to [47], composition can be realized by (i) *containment*, i.e., the specified transformation nests the reusable artifact, (ii) *connection*, i.e., the specified transformation reuses the artifact by delegation, (iii) *extension*, i.e., the reusable artifact is extended and refined, and (iv) *coordination*, i.e., a synchronization language is used to coordinate the reusable artifacts.

## 3 Comparison of reuse mechanisms

In the following, the identified reuse mechanisms, which are based on the findings of [44] and extracted from the field of software engineering, are compared along the five different scenarios of reuse. For each mechanism, first the *basic notion* is explained, second the mechanism is *exemplified*, and third, *conceptually evaluated* based on the introduced comparison framework. The results of this conceptual evaluation for each mechanism are summarized in Table 13. Finally, we investigate on the *realization* of the reuse mechanisms offered by dedicated M2M transformation languages or approaches introduced in literature.

To compare different realizations of reuse mechanisms, three orthogonal dimensions analogous to the three primary building blocks of programming languages [2] are investigated. The first two dimensions comprise static criteria: (i) the *syntax* a transformation language offers with respect to reuse mechanisms and (ii) the *static semantics*, which indicates whether the specification is well formed at design time or not. The third dimension of the comparison framework describes how reuse mechanisms behave at run-time, i.e., their *dynamic semantics*. The whole comparison framework as well as its relationships to reuse mechanisms and reuse scenarios is illustrated in Fig. 8.

### 3.1 Selected transformation languages and reuse mechanisms

In order to provide an extensive survey of reuse in dedicated M2M transformation languages, representatives of the three common paradigms of imperative, declarative, and hybrid transformation languages have been chosen [22]. The rationale behind this selection was to assort a representative mix of well-published approaches or standards and to keep the ratio between the different paradigms. Concerning imperative approaches, we considered Kermeta[1] (version 1.4.0) and the QVT Operations language of the QVT standard. Concerning declarative approaches, we investigated on the QVT Relations, being the declarative language of the QVT standard, TGGs,[2] VIATRA,[3], and our own transformation language called Transformation Nets [60]. As hybrid transformation languages, we considered ATL[4] (version 3.1.0), ETL[5] (version 0.9.1), and RubyTL[6] (version 3—alpha).

To identify reuse mechanisms that base on existing transformation languages (as often proposed for reuse scenarios 3–5), the methodology was to start from a comparison of reuse mechanisms in software engineering conducted in [44]. The identified mechanisms in this survey have been the starting point for our comparison, whereby we added more recent reuse mechanisms such as AOP as well as reuse mechanisms being specific to transformation languages such as HOTs by conducting a systematic literature review. First, we investigated the publication record of renowned authors of transformation languages in order to identify proposed reuse mechanisms. Second, we considered proceedings of conferences in the area model engineering, in particular those of MoDELS, ICMT, ECMFA, and their co-located workshops, as well as publications in journals, especially, SoSym.

### 3.2 Scenario 1: concrete intra-transformation reuse in the small

Mechanisms to avoid code duplication and thus to enhance readability and maintainability within a single transformation include *functions* and *rule inheritance*, since they depend on concrete metamodel types. Before we go into detail on these mechanisms, we first shortly introduce *code scavenging* for integrity reasons.

#### *3.2.1 Code scavenging*

Similar to traditional programming, transformation designers may adopt an ad hoc reuse mechanism by *scavenging* a reusable artifact from existing transformations and using them as part of their new transformation.

**Conceptual evaluation** Code scavenging does not support explicit means for *abstraction*. Instead, abstraction exists only in the mind of the transformation designer

[1] http://www.kermeta.org.

[2] http://www.moflon.org.

[3] http://www.eclipse.org/viatra2.

[4] http://www.eclipse.org/atl.

[5] http://www.eclipse.org/epsilon/doc/etl.

[6] http://rubytl.rubyforge.org.

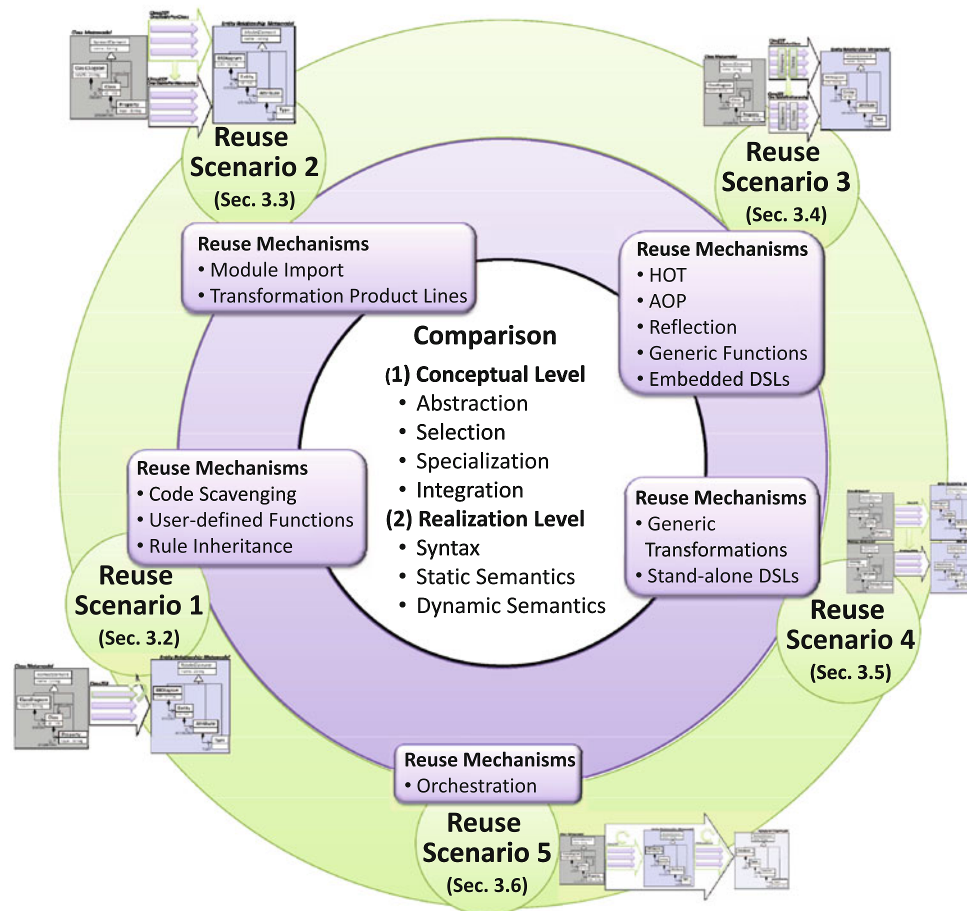


**Fig. 8** Overview on classification of reuse mechanisms and on applied comparison framework

[44]. Concerning *selection*, transformation designers typically recognize similarities between (parts of) transformations they currently design and (parts of) transformations they have previously defined, i.e., the selection process is purely manual. Therefore, the main repository for code scavenging is the own code. *Specialization* must be done manually by the transformation designer in a white-box manner. Since the reused artifacts are just copied into the new transformation, the composition kind is containment. Thus, although code scavenging is a reuse mechanism, which is possible in all transformation languages, its effectiveness is restricted by its informality. Therefore, no detailed comparison across different transformation languages is provided.

### 3.2.2 Functions

Functions provide means to extract and then reuse recurring logic. They are a well-known reuse mechanism from programming, and thus, have also been considered in numerous transformation languages.

*Example* Figure 9 shows an ATL helper function, realizing the concatenation of the name attribute with the string

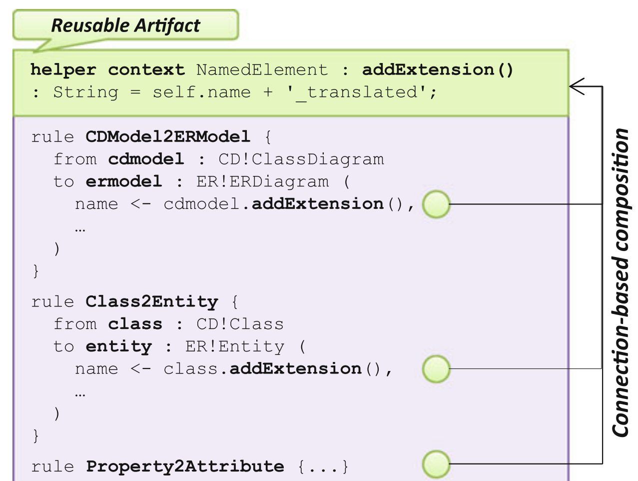


**Fig. 9** Example: functions in ATL

"_translated." This function is then invoked several times in the transformation specification.

**Conceptual evaluation** The gained *abstraction* of this reuse mechanism is low, since functions typically depend on con-

**Table 2** Realization of functions in M2M transformation languages

| Criteria | Values | Kermeta | QVT-O | QVT-R | ATL | ETL |
|---|---|---|---|---|---|---|
| *Syntax* | | | | | | |
| Context | ✓ (Supported)/ × (Not supported)/ | ✓ (MM as classes) | ✓ | × | ✓ | ✓ |
| Overloading | ✓ (Supported)/ × (Not supported)/ | × | ✓ | ✓ | ✓ (But not interpreted) | ✓ |
| *Static semantics* | | | | | | |
| Compatibility with declaration | [Compile-time/ run-time/no] error | Compile-time error | Compile-time error | Run-time error | Run-time error | Run-time error |
| Duplicate declarations | [Compile-time/ run-time/no] error | Compile-time error | Compile-time error | No error (first function selected) | Compile-time error | No error (first function selected) |
| *Dynamic semantics* | | | | | | |
| Dynamic binding | ✓ (Supported)/ × (Not supported)/ | ✓ (MM as classes) | ✓ | n.a. (no context) | ✓ (only on MM types) | ✓ |

crete metamodel types (cf. context `NamedElement` in Fig. 9). However, abstraction by simplification is gained, since the implementation is hidden. For *selection*, no publicly available repository exists, instead user-defined libraries may exist, only. *Specialization* is done black-box based, i.e., functions are specialized in a language-inherent manner by parameter values. Concerning *integration*, functions are a connection-based composition mechanism.

Numerous transformation languages provide support for functions. To achieve a representative set for comparison, we selected: (i) imperative languages, being Kermeta [69] and QVT-O [25], (ii) declarative languages, being QVT-R based on ModelMorf [66], and (iii) hybrid languages, being ATL [5] and ETL [70]. Although the concept of functions is well understood in general, functions in transformation languages are realized differently, as summarized in Table 2 and discussed in the following.

**Criteria for language evaluation** With respect to the *syntax of functions*, we investigated two criteria. First, a function may depend on a certain *context*, i.e., the function may be called for a certain type of element, only, which is typically an element of the source or target metamodel. Second, the possibility of *overloading* functions, i.e., functions exhibiting the same name, but a different parameter list, is explored.

Concerning the *static semantics* of functions, it should be ensured that the called function is *compatible* to its *declaration*, e.g., that a called function has been declared before and that the types and numbers of parameters are valid. Furthermore, *duplicate declarations* of functions should be detected at compile-time, i.e., equally named functions with the same parameter list.

Finally, regarding the *dynamic semantics*, it should be analyzed if functions support *dynamic binding*, i.e., functions should be called depending on their dynamic context type.

**Evaluation of languages** When investigating the *syntactical specification* of functions, it has to be noted that QVT Relations do not allow to specify a *context* for a function. In Kermeta, this criterion is only applicable if transformation rules as well as the metamodels are represented as classes and according functions as operations. Furthermore, ATL distinguishes between so-called functional helpers and attribute helpers. In contrast to functional helpers, which are considered in the following only, attribute helpers do not allow the specification of parameters. Such attribute helpers are often used to introduce derived attributes. *Overloading* is supported by the investigated transformation languages except Kermeta. However, in ATL, overloading is not interpreted, since parameters are not used to discriminate helpers that have the same name and the same context. This implies that all the helpers defined within a given context in an ATL program must have a distinct name. In case that two equally named helpers with the same context are defined, the last declared helper is called. A further peculiarity of ATL is the `super` keyword in functions which allows to call functions with the same name defined for a super type of the context type.

Regarding *static semantics*, only Kermeta and QVT-O evaluate if a function call is *compatible with its declaration* at compile-time. All other investigated languages throw a run-time error if the according function does not exist or exhibits a different parameter list. Concerning the detection of *duplicate declarations* of functions, Kermata, QVT-O, and ATL raise a compile-time error. In contrast, QVT-R and ETL do not raise any error, neither at compile-time nor at run-time. Instead, the first function specified in the source code is called.

With respect to the *dynamic semantics*, all languages which allow to specify a context of a function support *dynamic binding*, i.e., equally named functions exhibiting a different context are selected on basis of the dynamic type. However, ATL exhibits the constraint that dynamic binding is not supported for primitive types, whereas this is supported in QVT-O and ETL.

### 3.2.3 Rule inheritance

The concept of inheritance plays a major role in metamodels, as revealed, e.g., by the evolution of the UML standard [51]. In metamodels, it is common to employ inheritance between classes to reuse feature definitions from parent classes. Since classes and their respective objects are typically input or output for transformation rules, inheritance between transformation rules dealing with classes that are in an inheritance hierarchy in the metamodels may be applied in order to avoid code duplication.

*Example* Figure 10 shows an example of rule inheritance in ATL, where inheritance is used to avoid the re-specification of the `name` assignment by inheriting from the abstract transformation rule `NElem2MElem`.

**Conceptual evaluation** Rule inheritance does neither achieve *abstraction* from the actual metamodel types nor from the underlying transformation language. Furthermore, no abstraction by simplification takes place, since the whole implementation of the superrules is exposed to transformation designers. For *selection*, no repository exists since inheritance is currently specific to a single transformation. Superrules might be *specialized* by overriding them in a white-box, language-inherent manner. With respect to *integration*, inheritance represents an extension-based composition mechanism.

Although inheritance is a well understood and commonly agreed reuse mechanism in software engineering, current transformation languages supporting inheritance exhibit differences as illustrated by a selection of languages, comprising (i) the imperative languages Kermeta [69] and QVT-O [25], (ii) the declarative languages TGGs [41] and TNs [60], and (iii) the hybrid languages ATL [5] and ETL [70]. The results of the comparison are summarized in Table 3.

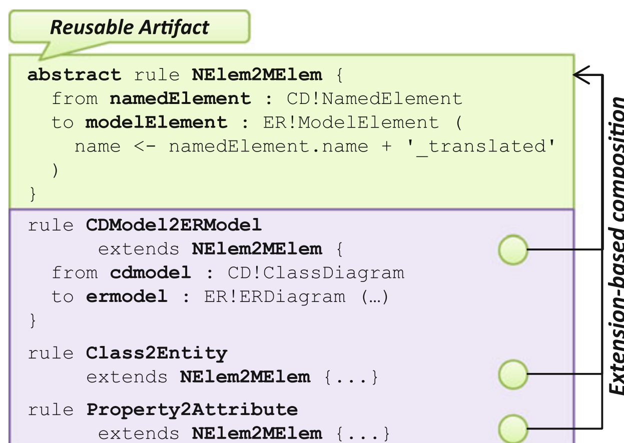


**Fig. 10** Example: rule inheritance in ATL

**Criteria for language evaluation** Concerning the *syntax of inheritance*, two criteria have been investigated. First, a transformation rule may inherit from either one or multiple other transformation rules, depending on whether *single or multiple inheritance* is supported. Second, the concept of *abstract* rules may be supported in order to specify that a certain rule is not executable per se but provides core behavior that may be reused in subrules.

With respect to the *static semantics of inheritance*, it must be statically ensured that subrules refine the types of the superrules in a *co-variant* manner, only. Second, *concrete rules targeting an abstract class* should result in a compile-time error, since abstract classes are not instantiable. Finally, in case of multiple inheritance, potentially arising *diamonds* should be statically checked.

Regarding the *dynamic semantics*, it is of interest, which rules apply to which instances, i.e., whether a transformation language supports the well-known principle of *type substitutability* [49]. In the context of model transformations, this means if no specific subrule is defined for instances of a subclass, then these instances of the subclass may be transformed by the rule defined for the superclass. Second, it is of interest, how inherited *conditions* are treated, i.e., whether inherited conditions are also evaluated in subrules (cf. composing behavior) or not (cf. asymmetric behavior). Finally, the same question is relevant for *assignments*, i.e., how inherited assignments are treated.

**Evaluation of languages** Concerning the *syntax of inheritance*, all the languages evaluated except ATL support *multiple inheritance*. Furthermore, *abstract rules* are possible in all languages.

With respect to the *static semantics*, one may see that checking the static semantics is still limited in transformation languages. Regarding *type changes*, Kermeta is most restrictive, since no type changes are allowed. In contrast, all the other languages allow for *co-variance* of input elements and output elements, which is typically ensured at compile-time—the only exceptions are ATL and ETL. With respect to *concrete rules targeting abstract classes*, most of the languages detect this not before run-time, except QVT-O and TNs. Finally, the *diamond problem* is statically detected in all languages that support multiple inheritance, except in QVT-O.

Regarding the *dynamic semantics*, one main difference is the application of *type substitutability* in the different languages as may be seen in Table 3. Whereas ATL, TGGs, and TNs provide support by default, ETL allows the transformation designer to interfere. However, type substitutability is interpreted differently in ETL anyhow, since a superrule still regards all instances irrespective of whether the instances have already been matched by subrules or not. Furthermore, the imperative languages QVT-O and Kermeta also allow

**Table 3** Realization of rule inheritance in M2M transformation languages

| Criteria | Values | Kermeta | QVT-O | TGGs | TNs | ATL | ETL |
|---|---|---|---|---|---|---|---|
| *Syntax* | | | | | | | |
| Type of inheritance | Single/multiple | Multiple | Multiple | Multiple | Multiple | Single | Multiple |
| Abstract rules | Yes/no | Yes | Yes | Yes | Yes | Yes | Yes |
| *Static semantics* | | | | | | | |
| Non-co-variant type change | [Compile-time/ run-time/no] error | n.a. (signature must not be changed) | Compile-time error | Compile-time error | Compile-time error | Run-time error | No error |
| Abstract target classes | [Compile-time/ run-time/no] error | Run-time error | Compile-time error | Run-time error (application fails) | Compile-time error | Run-time error | Run-time error |
| Diamond problem | [Compile-time/ run-time/no] error | Compile-time error | No error | Compile-time error | Compile-time error | n.a. (single inheritance) | Compile-time error |
| *Dynamic semantics* | | | | | | | |
| Type substitutability | Yes/no | n.a. (depends on rule calling order) | n.a. (depends on rule calling order) | Yes | Yes | Yes | User-definable |
| Condition semantics | Asymmetric/ composing | n.a. (determined by programmer) | Asymmetric | Composing | Composing | Composing | Composing |
| Assignment semantics | Asymmetric/ composing | n.a. (determined by programmer) | Composing | Composing | Composing | Composing | Composing |

the transformation designer to interfere, since the calling of rules is performed by the transformation designer. In addition to that *conditions* are evaluated by a composing completion of the lookup—the only exception thereof is QVT-O, adhering to an asymmetric approach. Finally, all evaluated transformation languages implement a composing behavior for *assignments*.

#### *3.2.4 Synopsis*

Functions as well as rule inheritance are both mechanisms to avoid code duplication within a single transformation, complementing each other. Functions allow to reuse arbitrary expressions and are supported by diverse languages. However, semantic differences, especially concerning overloading, arise. In contrast, rule inheritance reuses assignments and conditions provided that the metamodels incorporate inheritance relationships between meta-classes. Although inheritance is an important reuse mechanism in object-oriented programming, not all transformation languages support rule inheritance, or if they do, they offer different semantics as has been shown in detail in [83].

### 3.3 Scenario 2: concrete intra-transformation reuse in the large

Provided that a similar transformation scenario has to be realized on the basis of an existing transformation, i.e., a transformation between the *same source and target metamodels*, but with *different transformation logic*, mechanisms are needed that allow to either alter the existing transformation, e.g., by *module import*, or to configure an existing transformation such that it meets certain requirements, e.g., by *transformation product lines*.

#### *3.3.1 Module import*

Import of functionality is a well-known reuse mechanism in software engineering, since the introduction of the module concept, which has also been considered in numerous transformation languages to split up transformations into manageable size and scope [21,79]. Module import allows to build the union of transformation rules from different transformations. Thereby, rules or helpers may be eventually redefined, i.e., a rule or a function may be replaced by a new one, and additional rules and functions may be added.

*Example* For exemplifying module import, Fig. 11 provides an example in ATL, whereby ATL calls the module import mechanism superimposition. A new transformation implementing a “one table per hierarchy” strategy should base on an existing “one table per class” transformation. Thereby, the superimposed transformation redefines the rule `Class2Entity` and adds an additional helper `Closure` for calculating the transitive closure.

**Conceptual evaluation** Module import neither *abstracts* from the metamodels, since old and redefined rules are bound to concrete metamodel types, nor from the transformation language. Module import also does not abstract by simplification, since the whole original transformation is visible

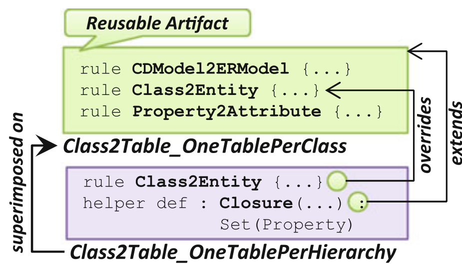


**Fig. 11** Example: superimposition in ATL

to the transformation designer. Concerning *selection*, any existing transformation could be reused in general. However, currently only the "ATL Model Transformation Zoo"[7] with publicly available transformations exists. Nevertheless, the selection process is supported by documentation, only. *Specialization* is done in a language-inherent, black-box manner, since redefining existing transformation rules has to be done in the same language and requires to know the exact signatures of the to be redefined transformation rules. Regarding *integration*, module import represents an extension-based composition mechanism.

In the following, we investigate module import support in the imperative transformation language QVT-O [25], the declarative transformation languages TGGs [41] and QVT-R based on ModelMorf [66], and the hybrid transformation languages ATL [5], ETL [70], and RubyTL [17]. The results may be found in Table 4.

**Criteria for language evaluation** With respect to the *syntax*, first we investigated, which *keywords* are offered for specifying the module import. Furthermore, the possibility for *redefining transformation rules* is investigated and how this may be achieved, e.g., either *implicitly*, e.g., by using equal rule names, or *explicitly*, e.g., in the form of specific keywords.

Concerning *static semantics*, it is evaluated whether static checks take place *across module boundaries*, e.g., if a certain transformation rule exists in one of the imported transformations. Furthermore, since transformation rules are bound to concrete source metamodel types, it should be prohibited to *change* the *source type*, since otherwise an additional rule is added rather than an existing one is changed. Concerning the output elements, it should be possible to change the according target type in order to alter the transformation behavior. Nevertheless, since other transformation rules may depend on the result of the to be refined rule, e.g., to set certain links, it has to be ensured that the *types* are *compatible*. In this respect, only co-variant changes, i.e., refined rules may return a subtype, should be allowed and *contravariant* type changes or a return type not contained in the inheritance hierarchy should be detected at compile-time.

Finally, regarding *dynamic semantics*, it is first investigated which "union" semantics is considered for *redefined transformation rules*. Thereby, *override* means that the new rule redefines the existing rule, whereas *generalization* means that the behavior of the redefined rule is kept, but additional logic may be added by a redefining rule. Finally, it is investigated whether *inheriting transformation rules* still behave correctly in case a superrule is refined, i.e., if module import or superimposition work properly in combination with inheritance.

**Evaluation of languages** All languages except ATL allow to import transformation definitions *statically* by means of explicit keywords. In QVT-O, the keyword `extends` is provided, in order to base a new transformation on an existing one. In TGGs, it is possible to *merge* the rule types, i.e., the high-level correspondences from one transformation with those of a new one. In QVT-R, it is possible to *import* a dependent transformation file and to *extend* a certain transformation of this file. Please note that QVT-R allows to specify more than one transformation per file, and thus, both keywords are required in order to identify the transformation to extend. ETL allows to *import* rules from a different transformation definition and so does RubyTL. In contrast, ATL provides the so-called superimposition mechanism, which builds a virtual transformation that contains the union of all transformation rules dynamically at load time [78]. The transformations that should be superimposed may be specified in the run-time configuration of an ATL transformation. With the exception of QVT-R, rules of the base transformation are *implicitly* redefined, if the imported transformation exhibits equally named transformation rules or as in RubyTL, traces for the source and target elements of a match are already available from the execution of the imported module. In contrast, in QVT-R, the redefinition of rules needs to be *explicitly* specified by means of the keyword `override`.

Concerning the evaluation of *static semantics*, those approaches that import transformations at compile-time provide means for static *cross-module checks*. In TGGs, this is not applicable, since currently there is no implementation available that supports merging of transformation rules—instead the evaluation is based on [41], but no details are mentioned regarding this point. Furthermore, QVT-O, TGGs, and QVT-R raise a compile-time error, if the *type of the source metamodel element* is changed in case a rule is redefined. In ATL and ETL, no error is raised—not even at run-time. Concerning a *contravariant change of output elements*, QVT-O and TGGs raise an according compile-time error. In QVT-R, this property is handled even more strictly, since no changes of the types of output elements are allowed at all. In contrast, ATL and ETL do not raise any error. Because the merging of rules in RubyTL is a purely dynamic approach based on refining already available traces, static semantic checks are not applicable.

[7] http://www.eclipse.org/m2m/atl/atlTransformations.

**Table 4** Realization of module import in M2M transformation languages

| Criteria | Values | QVT-O | TGGs | QVT-R | ATL | ETL | RubyTL |
|---|---|---|---|---|---|---|---|
| *Syntax* | | | | | | | |
| Import specification | Keyword | `extends` | `merge` | `import`, `extends` | Run configuration | `import` | `import` |
| Rule redefinition | Implicit/explicit | Implicit (equal names) | Implicit (equal names) | Explicit (keyword `overrides`) | Implicit (equal names) | Implicit (equal names) | Implicit (trace exists) |
| *Static semantics* | | | | | | | |
| Cross-module checks | [Compile-time/ run-time/no] error | Compile-time error | n.a. (no prototype) | Compile-time error | n.a. (load-time composition) | Compile-time error | n.a. (purely dynamic) |
| Change of source type | [Compile-time/ run-time/no] error | Compile-time error | Compile-time error | Compile-time error | No error | No error | n.a. (purely dynamic) |
| Contravariant change of output | [Compile-time/ run-time/no] error | Compile-time error | Compile-time error | Compile-time error (no change) | No error | No error | n.a. (purely dynamic) |
| *Dynamic semantics* | | | | | | | |
| Semantics of redefinition | Override/ generalization/ refinement | Override | Generalization | Override | Override | Override | Refinement |
| Rule inheritance | Broken/not broken | Broken (old base rule) | n.a. (no prototype) | n.a. (no inheritance) | Broken (old base rule) | n.a. (base rule may be substituted) | n.a. (no inheritance) |

Concerning the *dynamic semantics of the redefinition* of transformation rules, QVT-O, QVT-R, ATL, and ETL provide an *override* semantics, i.e., rules with the same name override the existing rule. Thereby, overriding means that the original rule is replaced with a new one, and it is not possible to refer to the original rule anymore. In contrast, TGGs impose a *generalization* semantics, i.e., the redefined TGG rule extends the original one. In RubyTL, rules are not composed, but refining rules are executed after the refined rules. In particular, refining rules only add additional bindings that are executed in addition to the bindings of the refined rules by following a similar check-before-enforce semantics as known from QVT Relations. Concerning the interplay between inheritance and module import, it has to be noted that in QVT-O, the rule inheritance hierarchy is broken, since inheriting rules still depend on the original base rule. Additionally, the refined base rule does not recognize that elements may have been matched by a more specific subrule, resulting in too many target elements. In TGGs, this criterion is not evaluated, since no prototype is available, and no information may be found on this aspect in [41]. Since QVT-R does not support rule inheritance, this criterion is not applicable. In ATL, superimposition is a load-time mechanism, whereas rule inheritance is flattened already at compile-time, i.e., transformation logic of the superrule is copied to the subrule during compilation. This leads to the fact that the inheritance hierarchy is broken, since a new base rule may not be considered by subrules anymore, because the overriding rule is not imposed before run-time. In ETL, this criterion is not evaluated, since type substitutability may be defined by the user (cf. previous subsection). Finally, RubyTL does not provide rule inheritance.

### *3.3.2 Transformation product lines*

Software product line engineering is a method for creating a collection of similar software systems [16]. Similar approaches recently gained attraction in the area of model transformations allowing to externally configure a transformation for easing its adaptation to slightly different scenarios. We call such a configurable family of model transformations *transformation product lines (TPLs)*. These approaches use a proper variability representation, e.g., in terms of a feature model [23], to guide the generation of a specific transformation.

*Example* Figure 12 shows a simplistic TPL for our running example, whereby a feature model allows to choose the classes to be translated as well as the applied object/relational mapping approach. From a specific selection in the feature model, the corresponding transformation satisfying the feature selection is produced.

**Conceptual evaluation** In TPLs, the reusable artifact is not only the already existing transformation, but also additionally the feature model, which models interdependencies and constraints of a model transformation. Since TPLs realize a set of related transformations, they are bound to concrete metamodel types and thus *abstract* neither from metamodels nor from the transformation language. Currently, no publicly

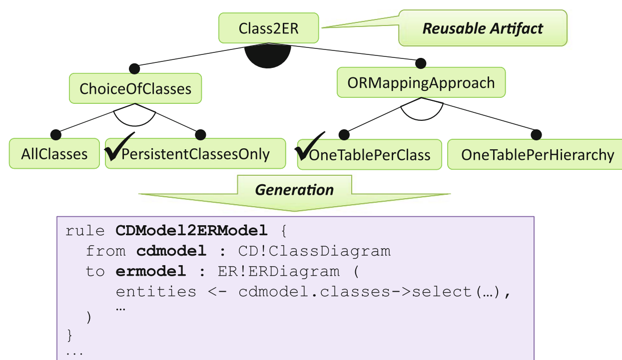


**Fig. 12** Example: transformation product lines in ATL

available TPLs may be *selected* from a repository. *Specialization* is done by configuring the feature model; thus, no internals of the transformation need to be known, being a black-box, non-language-inherent mechanism. Concerning *integration*, TPLs represent a generation-based reuse mechanism on basis of the configured feature model.

In the following, we investigate dedicated approaches that allow to realize TPLs. Since transformation product lines are a rather young research field in the area of model transformations, currently only two dedicated approaches are available (cf. Table 5). First, Kavimandan et al. [39] presented an approach called Model Transformation Templatization and Specialization (MTS) basing on the graph transformation language GReAT [7]. The second approach presented by Sijtema [63] introduces so-called variability rules on top of ATL.

**Criteria for language evaluation** First, *syntactic* means for the *specification of variabilities* are required. Thereby, we investigate if proprietary representations are employed or if the configuration is based on a standard feature model. Second, it is examined, if and with which means the *transformation needs to be adapted*, such that it becomes configurable.

Concerning the *static semantics*, *invalid selections in the variability model*, e.g., incomplete selections, should be statically prohibited. Second, it should be ensured that for all valid combinations in the feature model, transformation code is available, as represented by the criterion *inconsistencies between the variability model and the transformation*.

Regarding the *dynamic semantics*, the criterion *execution mode* examines if there is an interpreter that is able to execute the TPL or if the TPL specification is compiled to standard transformation code. Provided that compilation takes place, it is further investigated to which target language the variability model is translated to, i.e., the *underlying execution semantics*.

**Evaluation of languages** With respect to the *syntactic specification of variability*, MTS uses a *proprietary model*, whereby an existing model transformation may be annotated in order to identify variable parts by comments. Based on these comments, a variability metamodel is automatically extracted. In contrast, Sijtema bases on standard *feature models*. Concerning the specification of variability within the transformation, both approaches demand for *adaptations of the transformation*. In MTS, the transformation needs first to be annotated, in order to derive the variability metamodel, and second, it needs to be defined in a generic manner. The approach of Sijtema requires the specification of so-called variability rules that realize the different possible transformations configured by the feature model.

Concerning *static semantics*, both approaches do not ensure statically that a selection in the variability model represents a *valid combination*. Furthermore, none of the approaches checks whether the transformation covers the whole *variability model*, i.e., if there are *inconsistencies* between the transformation and the variability model.

Regarding *dynamic semantics*, both approaches rely on *compilation* by applying a *HOT* (cf. Sect. 3.4.1) to dedicated transformation languages. In MTS, the configuration of the variability model is used to instantiate a transformation in GReAT, thereby making use of the C++ template feature, to which GReAT is finally compiled to. In the approach of Sijtema, variability rules are transformed to standard ATL code, i.e., variability is translated to called rules in ATL, which refine a rule realizing a common base behavior.

### *3.3.3 Synopsis*

Both module import and TPLs allow to realize related transformation scenarios. Nevertheless, module import follows an ad hoc development approach, i.e., a transformation may be incrementally modified on demand, whereas TPLs represent a planned development approach, i.e., all potential variabilities of a transformation have to be modeled in advance. Although changes in TPLs themselves are challenging, since the feature model, the transformation code, as well as the code generator have to be adapted accordingly, TPLs have the advantage, that even domain experts without profound knowledge of a transformation language might configure transformations by just selecting values from the feature model. In contrast to TPLs, module import requires profound knowledge of the transformation language, but allows flexible changes of transformations.

## 3.4 Scenario 3: generic inter-transformation reuse in the small

Parts of transformation logic might be independent of any concrete scenario and might thus occur in various transformations, e.g., cross-cutting concerns like tracing or debugging. For the scenario of generic inter-transformation reuse in the small, *higher-order transformations*, *aspect-orientation*,

**Table 5** Realization of transformation product lines

| Criteria | Values | MTS | Sijtema |
|---|---|---|---|
| *Syntax* | | | |
| Variability model | Proprietary model/feature model | Proprietary model | Feature model |
| Transformation adaptation needed | Yes/no | Yes (comments) | Yes (variability rules) |
| *Static semantics* | | | |
| Invalid selection in variability model | [Compile-time/ run-time/no] error | No error | No error |
| Inconsistencies between variability model/transformation | [Compile-time/ run-time/no] error | No error | No error |
| *Dynamic semantics* | | | |
| Execution mode | Interpretation/ compilation | Compilation (HOT to GReAT) | Compilation (HOT to ATL) |
| Execution semantics | Target language | GReAT (C++) | ATL |

*reflection*, *generic functions*, and *embedded domain-specific languages* have been found in literature, which are characterized in the following.

#### *3.4.1 Higher-order transformations (HOTs)*

HOTs are model transformations that either take a model transformation as input, produce a model transformation as output, or do both. As detailed in [68], HOTs may be applied in several ways to achieve reuse in model transformations, being (i) *transformation composition*, (ii) *transformation synthesis*, and (iii) *transformation modification*. *Transformation composition* means that a HOT takes at least one transformation and potentially other configuration models as input and produces a transformation as output. Consequently, transformation composition may be used, e.g., to achieve genericity (cf. Sect. 3.5.1). *Transformation synthesis* implies that a transformation is generated from other artifacts. Thus, this type of HOTs is often applied in the context of domain-specific languages (DSLs) to generate transformations from DSL constructs (cf. Sects. 3.4.5, 3.5.2). Therefore, in this section, we focus on HOTs in the sense of *transformation modification*, e.g., a HOT takes a transformation as input to introduce cross-cutting concerns like debugging or tracing.

*Example* Figure 13 depicts a HOT developed with ATL that is able to add debugging messages to arbitrary ATL transformation rules. Consequently, the reusable artifact in terms of the HOT is independent of the concrete metamodels and may be reused across transformation boundaries.

**Conceptual evaluation** In the context of HOTs, the reusable artifact might be either the transformation, the introduced HOT or even both, depending on what is being newly developed. HOTs *abstract* from concrete metamodel types, but not from the underlying transformation language. Furthermore, they do not abstract by simplification, since no parts are explicitly hidden. Existing HOTS may be *selected* from a transformation repository, whereby several ATL-based HOTs are available in the ATL zoo. *Specialization* happens in a black-box manner, provided that only transformation-independent modifications take place, e.g., for each assignment add a debug message. The specialization mechanism is the HOT itself. Provided that the transformation to be specialized and the HOT itself are both written in the same transformation language, the HOT is considered to be language inherent. Concerning *integration*, HOTs form a composition-based reuse mechanisms in terms of extension.

**Evaluation of languages** HOTs are ordinary M2M transformations with the exception that the source and/or target metamodel is that of a transformation language [68]. Just like any model may be created, modified, or augmented through a model transformation, a model transformation itself may be created, modified, and augmented [67]. Consequently, ordinary transformation languages are used to develop HOTs. However, these transformation languages do not represent a dedicated reuse mechanism on their own—thus, languages for specifying HOTs have not been evaluated separately.

#### *3.4.2 Aspect-orientated transformations*

The idea of aspect-oriented software engineering is to provide proper means to separate concerns [40]. For this, aspect-orientation allows to weave code—denoted as *advices*—into different points inside existing software. All potential points,

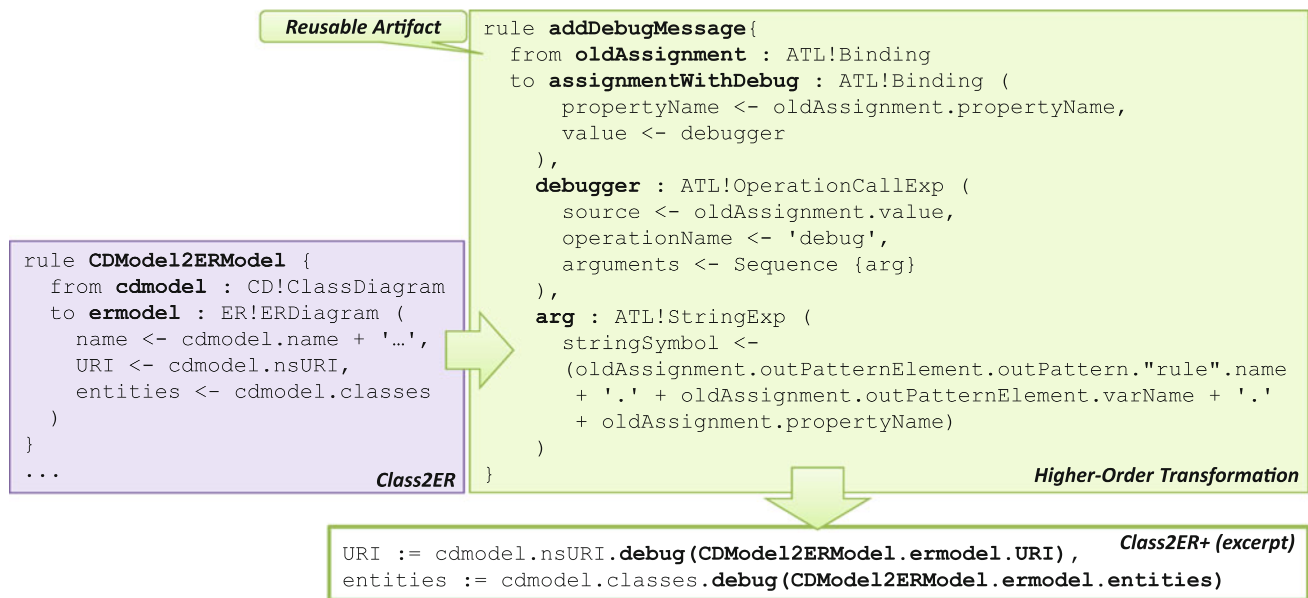


**Fig. 13** Example: higher-order transformation in ATL

where code may be weaved into, are called *join points*. Furthermore, *point cuts* allow to specify a kind of "query" on the potential join points, in order to select a subset of join points, where to actually weave advices into. This approach is summarized by the term *asymmetric aspect-orientation* [85]. In contrast, the symmetric aspect-orientated approach allows to assemble a program on the basis of equitable concerns, comparable to the reuse mechanism module import. The notion of aspects may also be adopted for model transformations as introduced in [57,76], e.g., to weave cross-cutting transformation code into existing transformations.

*Example* In Fig. 14, AOP is applied for separating the concern of printing debugging messages from a Kermeta transformation. Although this advice would be independent of concrete metamodels and may thus be reused across transformation boundaries in general, the realization in Kermeta is tightly bound to the class `ModelElem2Element`. This is, since a merge of concerns takes place, only, if the concerns in terms of classes are all equally named in Kermeta.

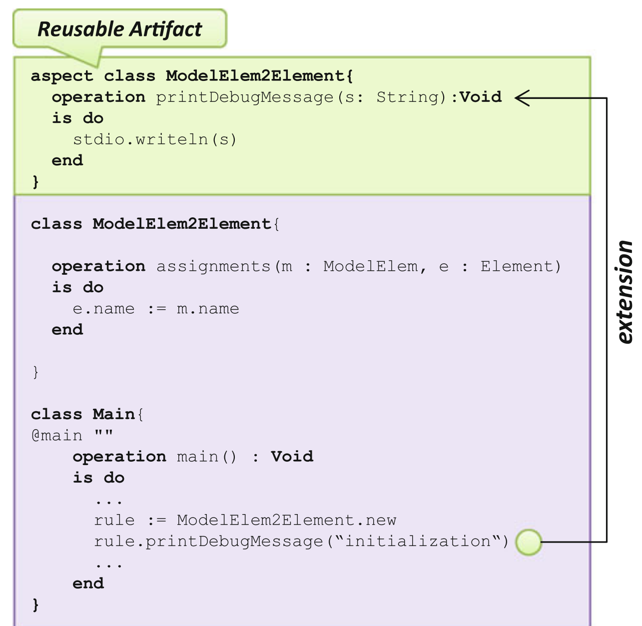


**Fig. 14** Example: AOP in Kermeta

**Conceptual evaluation** In aspect-oriented transformations, the reusable artifact is the code of the advice, if the advice represents a general transformation code that is applicable across transformation boundaries, e.g., debugging messages. Similar to HOTs, aspect-oriented transformations *abstract* from concrete metamodel types, but they do not abstract from the underlying transformation language, since aspect-orientation is a language-inherent feature. Furthermore, aspect-orientation does not abstract by simplification, since no parts are explicitly hidden. Although external advices could be easily reused, currently no repository for *selecting* aspects is available. *Specialization* happens typically as a black-box, e.g., for adding a debug message, no internals need to be known about the underlying transformation. The specialization mechanism is the language in which aspects and advices are defined as well as the join point model. Similar to HOTs, specialization is considered to be language inherent, if aspects and advice code are specified within the transformation itself. Concerning *integration*, aspect-oriented transformations form a composition-based reuse mechanisms in terms of *extension*.

**Evaluation of languages** To the best of our knowledge, the only M2M transformation language that currently supports aspect-orientation is Kermeta [69]. However, Kermeta provides symmetric aspect-orientation only, since no dedicated point cut model exists. Instead, classes may be marked by the keyword `aspect` allowing to contribute features (e.g., attributes or operations) to existing classes. As mentioned above, the assignment of the advice to the actual class representing the transformation rule is established by name equality. However, the semantics behind is a *merge*, i.e., all features of the aspect class and the base class are merged into a common class. Thereby, equally named features are overridden. Furthermore, overriding depends on the order in the file. In summary, this mechanism is rather comparable to module import than to traditional AOP, and therefore, we did not derive further specific criteria.

### *3.4.3 Reflection*

In object-oriented programming, reflection allows to examine and modify the structure and behavior of objects at runtime. Consequently, in the area of model transformations, reflection should allow to investigate (i) the involved metamodels (the structure) and (ii) the model transformation (the behavior).

*Example* Figure 15 shows an example of the reflective capabilities of Mistral. The target of the reflection is the behavior, i.e., the transformation. The provided example allows to add the production of trace information to arbitrary transformation rule executions, i.e., the addition of the trace concern is done at run-time.

**Conceptual evaluation** The reusable artifact in reflection is the reflective transformation. Reflective transformations *abstract* from concrete metamodel types, but not from the underlying transformation language, since reflective capabilities are typically tightly bound to a specific transformation language. Furthermore, reflection does not abstract by simplification, since no parts are explicitly hidden. With respect to *selection*, currently no dedicated repository of reflective transformations is available, although existing metarules could be reused in general. *Specialization* happens typically as a black-box, provided that transformation-independent modifications take place, only, e.g., adding a trace as shown in Fig. 15. The specialization mechanism is the reflective language itself, i.e., so-called metarules. Similar to HOTs, specialization is considered to be language inherent, if the reflection mechanism is integrated into the transformation language. Concerning *integration*, reflection forms a composition-based reuse mechanism in terms of extension.

Currently, reflection is explicitly supported by Mistral [46], a language on top of ATL, and TGGs in terms of the underlying language SDM (Story Driven Modeling) [48], as summarized in Table 6. Furthermore, it has to be noted that Kermeta, ETL, and RubyTL also support reflective techniques. While Kermeta has language inherent, object-oriented programming language like support for reflection, the latter two inherit this support from their respective host languages. In the following, we will focus on reflection mechanisms of Mistral and SDM, because they have been introduced specifically for transformation languages.

**Criteria for language evaluation** With respect to *syntax*, we examined first the *target of the reflection*, i.e., if syntactic means are provided to reflect on the *transformation*, on the *metamodels*, or on both. Second, the *information accessible* by reflection is investigated. In this respect, reflection may either access the static structure, i.e., *information* available at *compile-time*, or additionally access the dynamic behavior of the transformation, i.e., *run-time information*.

Concerning *static semantics*, no criteria have been investigated, since reflection is a mechanism that allows to query and change properties at run-time without dedicated assumptions at compile-time.

Finally, regarding *dynamic semantics*, two different *reflective capabilities* may be distinguished. First, reflection may enable *read access*, which is typically denoted as *introspection*. Second, reflection may also enable *write access*, e.g., to alter the transformation execution or metamodels at run-time, which is called *intercession*. Concerning the actual *execution mode* of the reflection mechanism, a *preprocessing* step may be introduced, e.g., a transformation language is syntactically extended, and the new constructs are translated to the existing languages' constructs by applying a HOT. Furthermore, the *interpreter or compiler* of the transformation language itself may have been extended to support reflection.

**Evaluation of languages** Regarding *syntax*, Mistral's reflective features exclusively *target* the *transformation* specification itself, whereas TGGs exclusively focus on the metamodels. Concerning the *accessible information* in the reflection process, Mistral supports *compile-time* as well as *run-time* reflection. To support run-time reflection, the transformation language metamodel is extended to represent the execution semantics of a transformation language. In contrast, TGGs only allow to access the information available at compile-time.

Concerning the *dynamic semantics of reflection*, Mistral offers the ability of *introspection* and *intercession*. For instance, it is possible to add an additional output parameter to a transformation rule at run-time (cf. Fig. 15). TGGs also support both mechanisms, e.g., it is possible to initialize all String-typed attributes of a certain class with the empty String by a single reflective rule. Concerning the *execution mode of reflection*, Mistral extends its *interpreter* by dedicated means for reflection. Since TGGs are compiled to Java, at least in

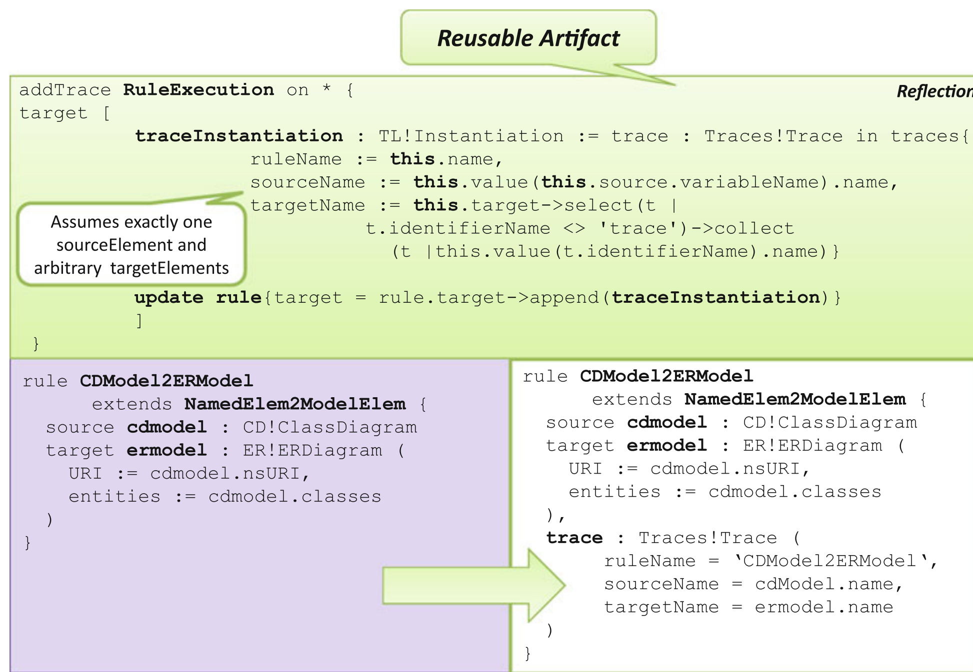


**Fig. 15** Example: run-time reflection in Mistral

**Table 6** Realization of reflection in M2M transformation languages

| Criteria | Values | Mistral | TGGs |
|---|---|---|---|
| *Syntax* | | | |
| Target of reflection | Metamodel/transformation | Transformation | Metamodel |
| Accessible information | Compile-time information/run-time information | Compile-time and run-time information | Compile-time information |
| *Dynamic semantics* | | | |
| Reflective capabilities | Introspection (read access)/intercession (write access) | Introspection and intercession | Introspection and intercession |
| Execution mode | Preprocessing/interpreter/compiler | Interpreter | Compiler |

case of MOFLON, the compiler has been extended, since Java already supports reflection.

### *3.4.4 Generic functions*

Genericity is a well-accepted concept to decouple strong typing from implementation logic, e.g., it allows to parameterize methods with types. The same idea is applied to model transformations by parameterizing transformation rules with types, such that the transformation logic becomes decoupled from concrete metamodel types.

*Example* Figure 16 shows a generic function in TGGs, realizing a simple equality comparison of an attribute value with a given value, i.e., it compares whether a parameterizable

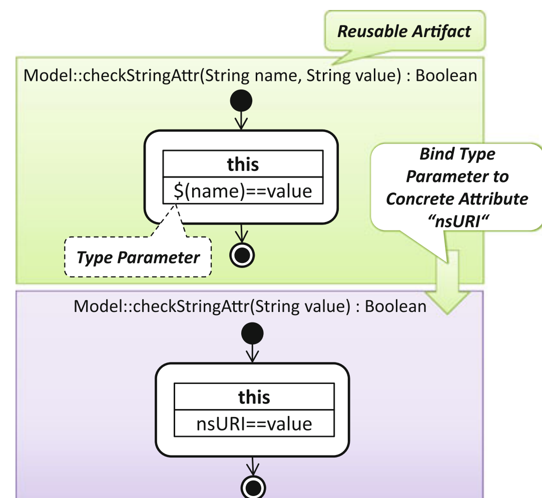


**Fig. 16** Example: generic function in TGGs

**Table 7** Realization of generic functions in M2M transformation languages

| Criteria | Values | Kermeta | TGGs | VIATRA |
|---|---|---|---|---|
| *Syntax* | | | | |
| Typing constraints on type parameters | Yes/no | Yes | No | Yes |
| Template specialization | Yes/no | No | No | No |
| *Static semantics* | | | | |
| Access to undeclared features | [Compile-time/ run-time/no] error | Compile-time error | Run-time error | Run-time error |
| Call of undeclared operations | [Compile-time/ run-time/no] error | Compile-time error | Run-time error | n.a. (no operations) |
| *Dynamic semantics* | | | | |
| Treatment of type parameter | Elimination/retention | Elimination (by type erasure) | Elimination (by type erasure) | Retention (native support) |
| Target language | – | Scala | Java | n.a (Interpreter) |

attribute exhibits a provided value, whereby type parameters are denoted by the “$” sign in TGGs.

**Conceptual evaluation** Generic functions *abstract* from concrete metamodels, but not from the underlying transformation language. The transformation designer only has to care about the source/target metamodels representing the visible part, whereas the implementation is hidden. Nevertheless, although the idea of generic functions is promising, no library for *selection* has been established so far, although generic functions would provide a high potential for reuse. Since *specialization* is done by setting type parameters, it is considered as black-box based. Finally, the specialization process occurs language inherent, and generic functions represent a connection-based composition mechanism for *integration*.

Although generic functions would be highly beneficial in the area of model transformations, little support is provided by current transformation languages. To the best of our knowledge, three transformation languages offer dedicated support, being Kermeta [69], TGGs [48], and VIATRA [73]. The results of the subsequent comparison are shown in Table 7.

**Criteria for language evaluation** Concerning the *syntax* of generic functions, we first examine whether it is possible to impose *typing constraints* on the *type parameters*. Furthermore, it is investigated whether it is possible to *specialize templates*—as known from C++. Template specialization allows—similar to method overloading—to specify several equally named templates, whereby one or more type parameters are specified using a concrete type. When instantiating a template, the most specific template is selected, automatically.

With respect to the *static semantics*, it should be ensured at compile-time that all *accessed features* of types are also declared, accordingly. Furthermore, the calling of undeclared operations should be statically recognized.

Finally, concerning the *dynamic semantics*, it shall be examined, how the *type parameters are substituted* by their concrete instantiation types and to which *target language* the transformation language is compiled to.

**Evaluation of languages** Concerning *syntax* of generic functions, Kermeta and VIATRA allow for *typing constraints* on generic functions. The situation is different in TGGs, since it is not possible to constrain the type parameters. Furthermore, all approaches do not allow to *specialize templates*. Concerning *static semantics*, Kermeta is able to statically detect, if certain features or operations accessed on a type parameter have also been declared, accordingly. In contrast, TGGs are not able to check this statically, since the type parameters are substituted at run-time. Thus, it is not possible to check whether a certain operation or a certain feature is present not before run-time. The same applies for VIATRA concerning features. Regarding *dynamic semantics*, the *replacement of type parameters* happens in Kermeta and VIATRA by *type erasure* [3], i.e., the compiler replaces all type parameters by the type `object` and introduces according type casts if needed, supported by the *underlying target languages*, i.e., Scala in case of Kermeta and Java in case of TGGs. VIATRA has native support by the accompanying interpreter where types are also represented as objects by applying multi-level metamodeling, i.e., also the instance of relationship between objects and types is a usual relationship.

### *3.4.5 Embedded domain-specific languages*

Another way to reuse logic in different scenarios are DSLs, which provide dedicated language constructs to simplify specification of recurring problems in transformations. In general, DSLs allow programs to be written at an abstraction

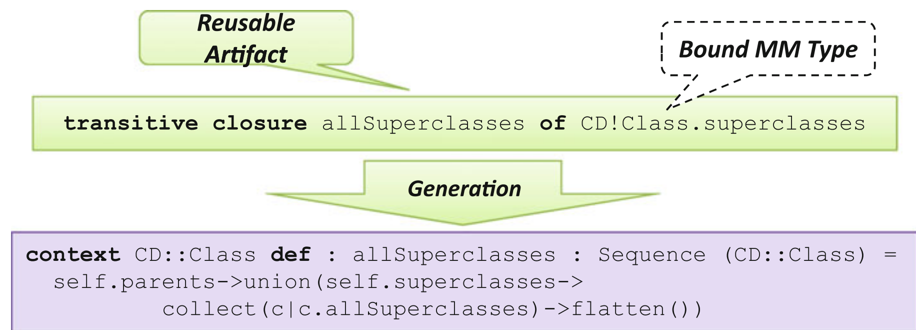


**Fig. 17** Example: embedded DSL in ATL

level closer to the problem domain than general-purpose programming languages [20]. In the area of model transformations, two dedicated kinds of DSLs have been proposed, comprising embedded DSLs and stand-alone DSLs,[8] whereby the former refers to the introduction of new language constructs in an already given transformation language, and the latter refers to the development of a DSL from scratch being independent from existing transformation languages. Consequently, DSLs demand for an additional compilation step, since the DSL constructs have to be either reduced to the original transformation languages' constructs in case of embedded DSLs or they have to be transformed to an executable form in case of stand-alone DSLs.

*Example* For exemplifying this category of reuse mechanisms, Fig. 17 depicts the proposed DSL construct "`allSuperclasses`" for ATL, stemming from the High-Level Navigation Language (HNL) [19]. It allows to calculate the transitive closure of all classes for a given inheritance hierarchy, whereby in the provided example "`CD!Class.superclasses`" has been bound to the DSL construct. Consequently, a transformation designer has to employ this simple DSL construct, only, instead of explicitly coding the underlying OCL query.

**Conceptual evaluation** Embedded DSLs provide *abstraction* from concrete metamodels, since the DSL constructs might be reused across metamodel boundaries, but not from the underlying transformation language, since the embedded DSL constructs are tightly coupled to a certain transformation language. Concerning *simplification*, the provided DSL syntax, i.e., the visible part, abstracts from the operational semantics, i.e., the hidden part. *Selection* of a certain reusable artifact, i.e., a DSL construct, is typically semi-automatically supported by editors, e.g., by means of code completion based on the DSL's grammar. DSLs are *specialized* in a black-box, language-inherent manner, since specialization is done by binding a certain grammar element to metamodel types. Since DSL constructs are compiled to ordinary transformation code, generation-based *integration* takes place.

As two representatives for embedded DSLs, we found the HNL [19], which hides complex OCL navigation expressions and ATL4pros [56], which eases the handling of UML profiles in ATL transformations. Since we refer to dedicated extensions of existing model transformation languages, only, we do not consider approaches that have used general-purpose languages to create new M2M transformation languages as internal DSL, e.g., RubyTL [20] (based on Ruby) or an ATL dialect implemented in Scala [28]. The results of our comparison are summarized in Table 8.

**Criteria for language evaluation** Concerning *syntax*, we will first examine the purpose of the DSL. In this respect, DSLs may be divided along four different kinds of *purposes* being (i) *generalization*, (ii) *compression*, (iii) *representation*, and (iv) *abstraction* [34]. In this context, *generalization* aims at reducing the amount of concepts, e.g., by building generalization hierarchies. The purpose of *compression* is to provide a concise language that is less verbose and easier to read, e.g., to reduce the amount of expressions without changing the semantics. DSLs, whose main purpose is on *representation*, focus on the provision of notations and abstractions that are best suited for a certain domain. Finally, the purpose *abstraction* summarizes DSLs, whose focus is on making common assumptions about a domain, which are kept transparent from the developer to deemphasize technical details. Second, it is of interest how the embedded DSL is *specified*. This may be either done by *extending the metamodel of the transformation language* itself or by *defining a separate metamodel*.

With respect to *static semantics*, it is of interest if the bound parameters to the DSL constructs are correct in type and number, i.e., if *type checking* takes place. Furthermore, it is examined if the embedded DSLs allow for *validation*, i.e., if the DSL constructs are syntactically correctly encoded.

Finally, concerning *dynamic semantics*, first, the actual *mechanism* applied is investigated, i.e., either if there exists an extension in the compiler or interpreter of the transforma-

[8] We refrain to speak about internal versus external DSLs, because this dichotomy reflects if a DSL is implemented by using the host language in which it is used or not. In contrast, we are only considering if the DSL is intended to be usable within a host language or not, independent of how it is implemented.

**Table 8** Realization of embedded DSLs

| Criteria | Values | HNL | ATL4pros |
|---|---|---|---|
| *Syntax* | | | |
| Purpose | [Generalization/compression/ representation/abstraction] | Compression | Compression |
| DSL specification | MM extension/ separate MM | Separate MM | MM extension |
| *Static semantics* | | | |
| Type checking of parameters | [Compile-time/ run-time/no] error | Run-time error | Run-time error |
| DSL validation | [Compile-time/ run-time/no] error | Compile-time error | Compile-time error |
| *Dynamic semantics* | | | |
| Mechanism | HOT/compiler | Compiler | HOT (in place) |
| Host language | – | ATL/OCL | ATL |

tion language or if a HOT is used to replace the specifications in the DSL solely by constructs of the host language. Second, the *host language* of the embedded DSL is examined.

**Evaluation of languages** With respect to *syntax*, the *purpose* of both embedded DSLs evaluated is to provide more simple language constructs, i.e., *compression*. In order to specify the HNL language features, a *separate metamodel* has been used, whereas in ATL4pros, the ATL metamodel has been *extended*, accordingly.

Regarding the *static semantics*, none of the DSLs evaluated provide *type checking of parameters*, but they both support *validation* of the DSL constructs.

Finally, concerning *dynamic semantics*, the *mechanism* for executing HNL statements is to accordingly extend the compiler. In contrast, ATL4pros applies a HOT to convert the DSL language constructs into standard ATL transformation code by introducing imperative `do`-Blocks. Thereby, the implemented HOT makes use of the in place mode of ATL in order to keep the HOT transformation as concise as possible. Second, regarding the *host language*, both approaches rely on ATL.

#### *3.4.6 Synopsis*

Although the first three reuse mechanisms of this scenario pursue similar goals, i.e., introducing additional transformation logic into existing transformations in a non-intrusive way, the main difference lies in the kind of specialization. Since HOTs are defined on the abstract syntax of a transformation language, a transformation designer must have profound knowledge thereof. In contrast, aspect-orientation allows specialization on basis of the concrete syntax (offered for defining the point cuts and advices) and reflection on basis of the provided language extensions. Generic functions are suitable to specify recurring transformation logic that should be reusable irrespective of the used metamodels. Finally, concerning embedded DSLs, they aim at similar goals as generic functions, but allow to introduce syntactic sugar by providing additional keywords.

### 3.5 Scenario 4: generic inter-transformation reuse in the large

Assuming that the same transformation logic should be reused in a different scenario, i.e., different source/target metamodels, mechanisms to decouple transformation logic from concrete metamodel types are needed. In this respect, *generic transformations* and stand-alone DSLs have been proposed as detailed in the following.

#### *3.5.1 Generic transformations*

Such transformations allow to parameterize transformation logic with types. However, in generic transformations not only functions should be decoupled from concrete type information, but also the transformations themselves to reuse a transformation by binding elements to different metamodels.

*Example* Figure 18 shows an example, whereby the whole `Class2ER` transformation should be reused for an `Ontology2XML` transformation. This is possible, since the metamodels of the new transformation are structurally similar to the metamodels of the existing transformation. For applying generic transformations, the transformation designer has to specify a binding model, which states, which generic types should be replaced by concrete types, e.g., `NamedElement` is replaced by `MElement`. The actual replacement of types is performed by a dedicated HOT [84].

**Conceptual evaluation** Generic transformations *abstract* from concrete metamodels. To achieve this for whole transformations, two kinds of approaches have been proposed. First, an *extrinsic binding* may be used by means of a binding model in case that the underlying transformation language does not support genericity or second an *intrinsic binding*

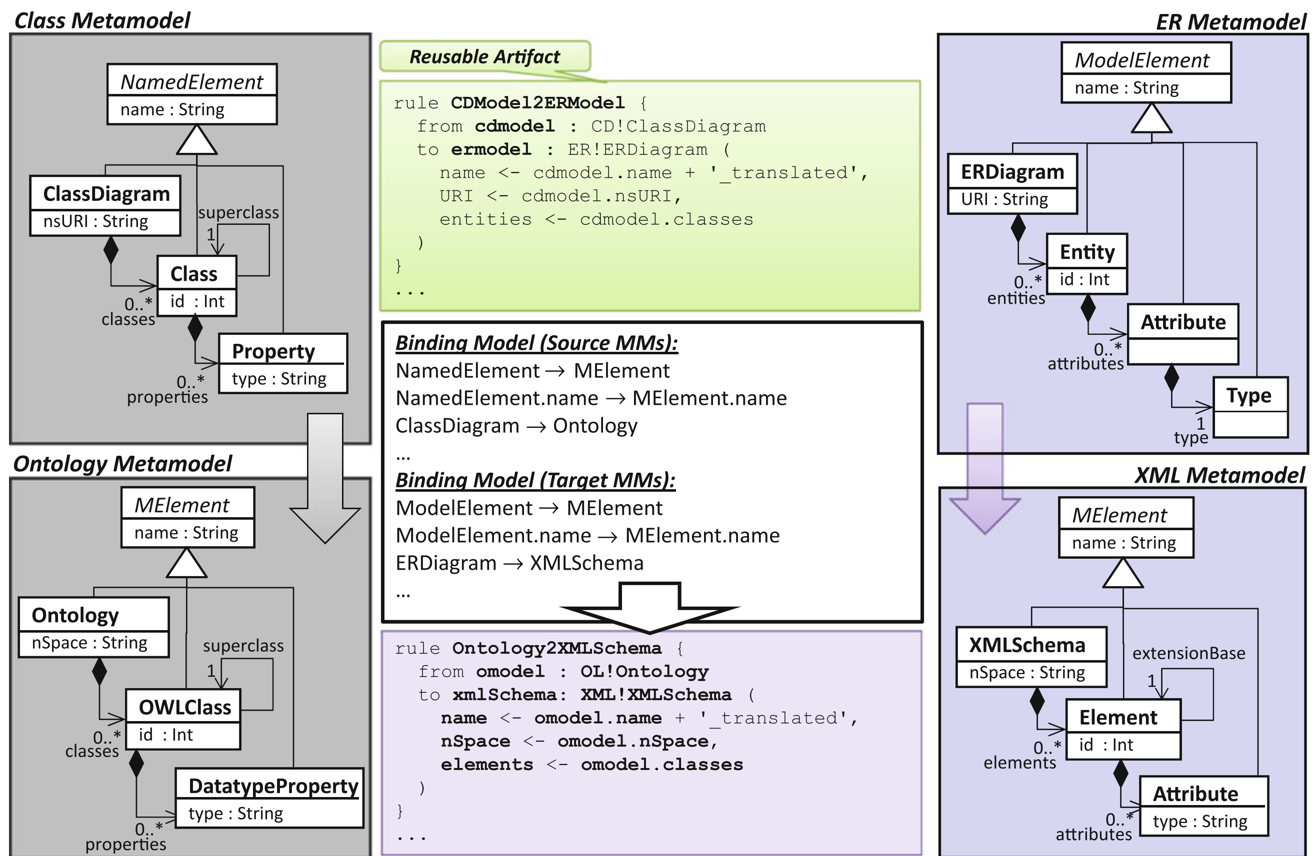


**Fig. 18** Example: generic transformation in ATL

in case that the underlying transformation language allows for generic rules. However, in both cases, the transformation designer has to take care about the source/target metamodels, only, representing the visible part, whereas the implementation is always hidden. Up to now, no library to *select* generic transformations has been established. Since *specialization* is done by specifying an extrinsic binding model or by binding parameters to generic functions, this reuse mechanism is considered as being black-box based. Finally, the specialization process occurs either non-language inherent in case of extrinsic binding models or language inherent in case of intrinsic bindings. Concerning *integration*, again two cases are possible: in case of extrinsic bindings, a generation-based integration takes place, whereas in case of intrinsic bindings, a composition-based integration is applied.

When investigating literature with respect to generic transformations, three approaches have been proposed, including [54,74,84], whereby the former two rely on intrinsic bindings and the latter one relies on extrinsic bindings. In the following, a detailed comparison is conducted.

**Criteria for language evaluation** Concerning the *syntax of generic transformations*, five main criteria have been investigated. First, the *realizable transformations*, i.e., the transformations that instantiate the generic transformations, may be *metamodel dependent*, i.e., the generic transformation makes assumptions about the structures of the metamodels by so-called concept metamodels [18]. To replace a concept metamodel by a specific metamodel, each element of the concept metamodel must be bound to an element of the specific metamodel as shown in Fig. 18. This approach is applicable, only, if a so-called subtype relationship between the concept metamodel and the specific metamodel might be established according to [64]. In contrast, the generic transformation may be *metamodel independent*, i.e., the transformation does not depend on the structures specified in the metamodels at all. Second, the *generic parts* are either the *source metamodel*, the *target metamodel*, or both. Third, the language for the *transformation template specification* is of interest. After having specified a transformation template, the *binding mechanism to concrete metamodels* is important, representing the fourth criterion. The final criterion refers to *support for resolution* of heterogeneities between the concept metamodel and the specific metamodel, i.e., if occurring heterogeneities need to be resolved manually or not.

With respect to the *static semantics of generic transformations*, two criteria have been employed. First, the ques-

tion arises if *missing bindings* are detected, thus checking whether all type parameters of the concept metamodel are bound to concrete types in the specific metamodel. The second criterion investigates on whether the bound parameters are statically checked for *type errors*, i.e., if wrongly typed parameters are detected at compile-time.

Regarding *dynamic semantics*, it is first investigated how the replacement of the type parameters is performed: either by means of a *pre-compiler* or by means of *type erasure* [3]. Second, the underlying *target transformation language* is of interest.

**Evaluation of languages** Concerning the *syntax of generic transformations*, the two approaches of Moha et al. [54] and Wimmer et al. [84] are *metamodel dependent*, i.e., make assumptions about the structure of the metamodels, whereas the approach of Varro et al. [74] does not. This is since the approach of Varro et al. introduces reflective graph patterns, which may be applied to all possible instances of metamodels irrespective of the concrete metamodel structure. For instance, such a graph pattern matches for all instances of arbitrary classes. Consequently, this approach is quite general, but suitable for very specific transformations, only, e.g., the translation of MOF-based metamodels into XMI. Concerning the *substitutable entities*, i.e., those metamodels, which might be exchanged, the approaches of Moha et al. and of Wimmer et al. allow to exchange both the source metamodel and the target metamodel. In contrast, the approach of Varro et al. allows for the exchange of the source metamodel, only. For *specifying the transformation template*, Moha et al. rely on generic Kermeta methods, Varro et al. on reflective VIATRA graph patterns, and Wimmer et al. on standard ATL transformations, which are rewritten in the binding process. The *binding of the transformation templates to concrete metamodels* is done by parameter binding in Kermeta by Moha et al. [54], by automatically binding all possible metamodel classes with a so-called metatransformation in VIATRA (a metatransformation has only one template type to which all metamodel classes are bound implicitly, thus a metatransformation may be seen as a special case of generic transformations) or by an explicit binding model in Wimmer et al. [84]. Concerning the *resolution support* of heterogeneities between the concept metamodel and the specific metamodel, Moha et al. do not provide any support. Instead, they propose resolution by manually specifying aspects in Kermeta. In contrast, the approach of Wimmer et al. provides dedicated resolution support for common heterogeneities. The transformation designer specifies one-to-one bindings between elements of the concept metamodel and the specific metamodel. Subsequently, the code for establishing the subtype relationship is automatically derived from these bindings. Regarding the approach of Varro et al., heterogeneities do not play a role, since the approach is metamodel independent.

With respect to the *static semantics*, the problem of *missing bindings* is recognized at compile-time by Moha et al., whereas Wimmer et al. do not detect this before run-time. Varro et al. are not confronted with missing bindings, since the type parameters in the reflective patters are automatically substituted with all available concrete types. The same evaluation results are also valid for *type checking* of the bindings.

Finally, regarding the *dynamic semantics* of generic transformations, the *replacement of the type parameters* is typically performed by a precompiler, whereby in case of Kermeta this step is delegated to Scala, where native support for generic methods is offered, basing on the concept of type erasure. Finally, Moha et al. base on Kermeta as target transformation language, Varro et al. on VIATRA, and Wimmer et al. on ATL (cf. Table 9).

### *3.5.2 Stand-alone domain-specific languages*

As embedded DSLs provide dedicated language constructs to simplify specification of recurring transformation logic, *stand-alone* DSLs provide the same functionality, but may be used independently from the underlying transformation language. In order to execute a DSL-based specification, it may either be translated into a certain executable transformation language or interpreted directly.

*Example* For providing an example, Fig. 19 shows a solution for the `Class2ER` transformation with the stand-alone DSL denoted as mapping operators (MOps) [80]. One may see that dedicated reusable components have been employed to describe the transformation logic such as the `VPartitioner` component, which allows to vertically split the instances of one class into instances of several classes. The reusable components ease the specification of recurring transformation scenarios. By a dedicated generation step, e.g., ATL code may be generated from the DSL specification (Table 10).

**Conceptual evaluation** Since being independent of a transformation language, stand-alone DSLs *abstract* from both the concrete metamodels and the underlying transformation language. Concerning *simplification*, the DSL syntax represents the visible part and abstracts from the operational semantics, i.e., the hidden part. *Selection* of a certain reusable artifact, i.e., a DSL construct, is typically semi-automatically supported by editors, e.g., by means of code completion based on the DSLs' grammar. DSLs are *specialized* in a black-box, language-inherent manner, since specialization is done by binding a certain grammar element to metamodel types, e.g., so-called ports need to be bound to a certain metamodel element in [80] (cf.

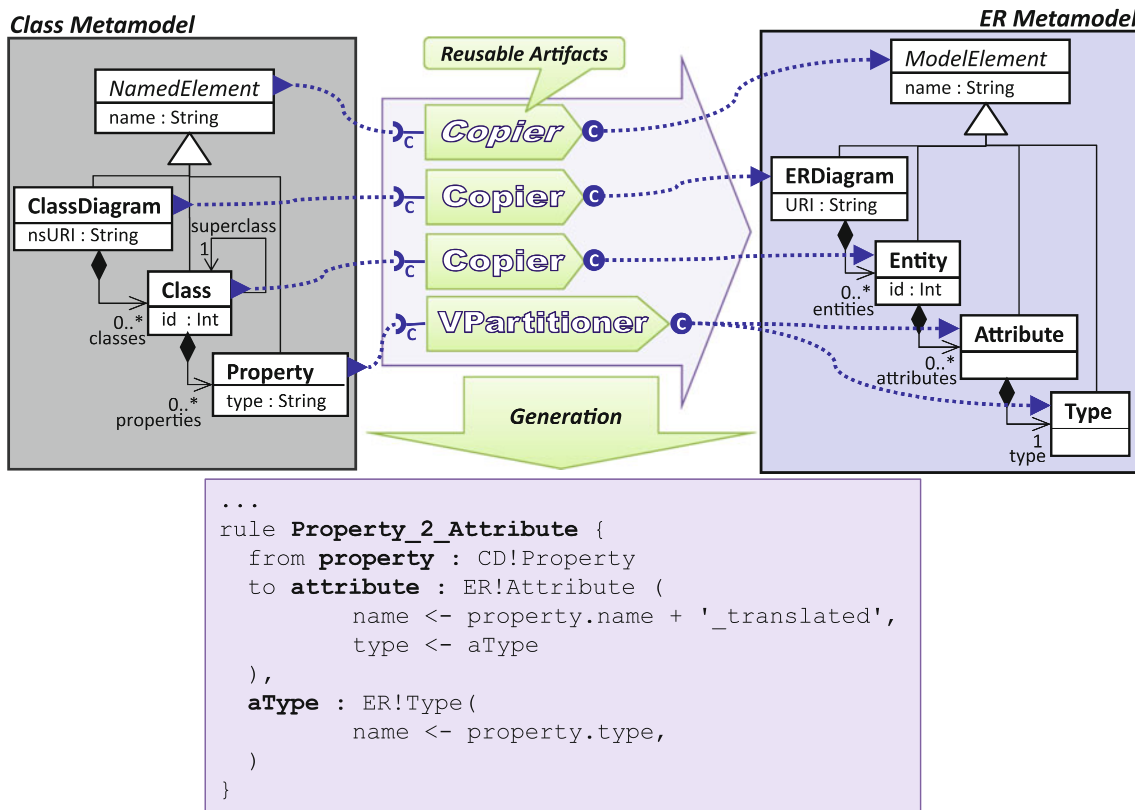


**Fig. 19** Example: stand-alone DSL [80] compiled to ATL

**Table 9** Realization of generic transformation approaches

| Criteria | Values | Moha et al. | VIATRA | Wimmer et al. |
|---|---|---|---|---|
| *Syntax* | | | | |
| Realizable transformations | MM-independent/ MM-dependent | MM-dependent | MM-independent | MM-dependent |
| Substitutable entities | Source MM/target MM/source/target MMs | Source/target MMs | Source MM | Source/target MM |
| Transformation template specification | Applied mechanism | Generic Kermeta operations | Reflective VIATRA graph patterns | Standard ATL transformations |
| Template binding | Applied mechanism | Parameter binding of generic operations | Automatically all model elements are bound | Binding model |
| Resolution support | Manual/semi-automatic/automatic | Manual | n.a. (arbitrary source MMs supported) | Automatic |
| *Static semantics* | | | | |
| Missing bindings | [Compile-time/ run-time/no] error | Compile-time error (Kermeta) | n.a. (automatic binding) | Run-time error |
| Type checking of bindings | [Compile-time/ run-time/no] error | Compile-time error (Kermeta) | n.a. (arbitrary source MMs) | Run-time error |
| *Dynamic semantics* | | | | |
| Replacement of type parameter | Precompiler/type erasure | Type erasure (compilation to Scala) | Precompiler (metatransformations) | Precompiler (HOT) |
| Target language | – | Kermeta | VIATRA | ATL |

Fig. 19). Since DSL constructs are typically compiled to ordinary transformation code, generation-based *integration* takes place.

Two dedicated stand-alone DSLs for specifying model transformations have been proposed, comprising AMW [24] and MOps [80], which are compared in the following with

**Table 10** Realization of stand-alone DSL

| Criteria | Values | AMW | MOps |
|---|---|---|---|
| *Syntax* | | | |
| Formalism | Textual/graphical | Graphical | Graphical |
| Composite DSL constructs | ✓ (Supported)/ × (Not supported)/ ? (Unknown) | ✓ | ✓ |
| DSL extensibility | ✓ (Supported)/ × (Not supported)/ ? (Unknown) | ✓ (Heavy-weight) | ✓ (Light-weight) |
| Expressivity | | | |
| Copying | ✓ (Supported)/ ∼ (Partly supported)/ × (Not supported)/ | ✓ | ✓ |
| Generating | ✓ (Supported)/ ∼ (Partly supported)/ × (Not supported) | ∼ | ✓ |
| Merging | ✓ (Supported)/ ∼ (Partly supported)/ × (Not supported) | ∼ | ✓ |
| *Static semantics* | | | |
| Type checking | [Compile-time/run-time/no] error | Compile-time error | Compile-time error |
| Validation | [Compile-time/run-time/no] error | No error | Compile-time error (inheritance checks) |
| *Dynamic semantics* | | | |
| Execution mode | Compilation/interpretation | Compilation (HOT) | Compilation (HOT) |
| Compilation to several target languages | Yes/no | No (ATL) | Yes (ATL, TNs) |

dedicated evaluation criteria. The results of the comparison are summarized in Table 10.

**Criteria for language evaluation** Concerning the *syntax of stand-alone DSLs*, four main criteria have been investigated. First, the DSL's syntax, i.e., the *formalism* for specification, may either be *textual* or be *graphical*. Second, a DSL may not only allow for flat DSL constructs, but also for *composite DSL constructs*, i.e., nesting of DSL constructs for hiding complexity of a model transformation may be supported or not. Third, a dedicated set of DSL constructs may be allowed to be *extended* by user-defined ones or not, whereby one may distinguish between *light-weight extensibility*, i.e., extensibility without the need to adapt the compiler/interpreter and *heavyweight extensibility*, i.e., extensibility, which entails the adjustment of the compiler/interpreter. Finally, the *expressivity* for specifying transformations on basis of DSL constructs is of interest. For evaluating this, the support for the three major transformation primitives of *copying*, *generating*, and *merging* has been investigated.

With respect to *static semantics of stand-alone DSLs*, two criteria have been employed. The first criterion refers to the possibility of *type checking*, i.e., whether the DSL ensures statically that only correctly typed metamodel elements are bound to the DSL constructs. Second, the possibility for *validation* is evaluated, i.e., whether the structure of the DSL constructs is validated according to certain criteria, e.g., if DSL constructs build a valid inheritance hierarchy.

Regarding *dynamic semantics*, again two criteria are of interest. First, it is checked whether the *execution* of the DSL constructs is done by *compilation*, i.e., by compiling the DSL constructs in another executable language, or by *interpretation*, i.e., if the DSL constructs are directly interpreted by, e.g., some virtual machine. Second, provided that a compilation-based approach is followed, the question arises whether the DSL constructs may be compiled into *several different target languages* or a single one, only.

**Evaluation of languages** Regarding the *syntax of the two DSLs* investigated, both approaches provide a *graphical formalism* for specification, allowing to *hide complexity by composition*. Concerning the *extensibility* of the languages by new DSL constructs, AMW provides a heavy-weight approach, only. In contrast, MOps employ a light-weight approach, since MOps rely on a set of primitive DSL constructs, denoted as kernel MOps, which are used in the compilation process. Consequently, arbitrary new composite MOps, which solely rely on kernel MOps, may be added, without the need of adjusting the compiler. Finally, regarding *expressivity*, only MOps provide an extensive set of DSL

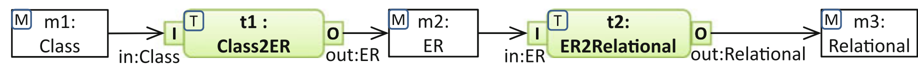


**Fig. 20** Simple orchestration example in Wires*

constructs, since all three transformation primitives of copying, generating, and merging are supported.

With respect to *static semantics*, one may see that *type checking* is provided in both DSLs at compile-time. In contrast, *static validation* of the DSL specification is provided by MOps, only.

Finally, regarding *dynamic semantics* of DSLs, both of them rely on *compilation*, whereby the approach of AMW is compiled into ATL, only. In contrast, MOps allow for a compilation of the DSL specification to ATL and to TNs [60].

### *3.5.3 Synopsis*

Generic transformations as well as stand-alone DSLs allow to decouple transformation logic from concrete metamodel types. The former are a promising approach to reuse transformation logic for structurally similar metamodels. Although large parts of transformation logic are reusable, the drawback is that they rely on structural similarity, resulting in a lower probability for application. In contrast, DSL constructs usually abstract from structural similarity to a certain extent, e.g., in [80] structural flexibility is supported by providing fixed parts as well as configurable parts. Consequently, DSLs may have a higher probability for application but do not provide transformations out of the box.

## 3.6 Scenario 5: concrete inter-transformation reuse in the large

To achieve reuse in the large, a set of transformations may be reused by consecutive execution. Thus, means to orchestrate model transformations are needed, e.g., describing sequential or conditional executions of M2M transformations (control flow) and how the models are streamed through the net of transformations (data flow).

### *3.6.1 Orchestration*

Orchestration languages have been proposed to replace low-level descriptions, e.g., Ant[9] tasks, to execute chains of transformations.

*Example* Figure 20 shows an example of orchestration in Wires* [58], specifying that the two ATL model transformations `Class2ER` and `ER2Relational` should be executed sequentially. Thereby, the output of the first transformation is used as input for the second one.

**Conceptual evaluation** Through the usage of orchestration languages, no *abstraction* from metamodels is achieved, since the transformations to be reused still operate on concrete metamodels. Also, no abstraction from the underlying transformation language is achieved, except the orchestration allows for transformations written in different languages. Concerning simplification, the hidden parts comprise the implementation of the transformation, since for orchestration the source and target metamodels of the transformations are of interest, only. In general, any existing transformation may be *selected*, however, currently only ATL transformations might be selected from the ATL zoo. Since transformations may be reused without adaptations, no *specialization* must occur. *Integration* happens by means of the orchestration language; thus, it is classified as coordination.

Orchestration of model transformations is supported by several approaches. In the following, we compare the orchestration languages Wires* [58], UniTI [72], the Transformation Composition Modeling Framework [55], MCC [42], ATLflow,[10] QVT-O, and RubyTL. The results are summarized in Table 11.

**Criteria for language evaluation** Concerning the *syntax of orchestration mechanisms*, five main criteria have been investigated. First, the *specification* of the transformation orchestration may be either *intrinsic*, i.e., a transformation language itself offers language constructs to orchestrate several model transformations or *extrinsic*, i.e., a separate formalism (e.g., activity diagrams) is employed for describing orchestrations. Furthermore, the employed *formalism* may be either *textual* or *graphical*, representing the second criterion. Third, the orchestration formalism may allow to orchestrate transformations written in *different languages* or the same language only, which is meant by *cross-language support*. Fourth, the possibility to *persist intermediate results* of the transformation chain is considered. Finally, the *expressivity* of the orchestration language has been investigated, checking whether the language allows for *nesting* for hiding complexity in transformation chains, *sequential composition*, *conditional composition*, *repetitive composition*, and *parallel composition*.

With respect to *static semantics of orchestration*, two criteria have been employed. First, before executing a specified model transformation chain, it should be ensured that the provided models are valid instances of the provided meta-

[9] http://ant.apache.org/.

[10] http://opensource.urszeidler.de/ATLflow/.

**Table 11** Realization of transformation orchestration approaches

| Criteria | Values | Wires* | UniTI | Transformation composition modeling framework | MCC | ATLflow | QVT-O | RubyTL |
|---|---|---|---|---|---|---|---|---|
| *Syntax* | | | | | | | | |
| Specification | Intrinsic/extrinsic | Extrinsic | Extrinsic | Extrinsic | Extrinsic | Extrinsic | Intrinsic (`access` keyword) | Intrinsic (`execute` keyword) |
| Formalism | Textual/graphical | Graphical (based on UML activities) | Graphical (tree editor of EMF) | Graphical (based on UML activities) | Textual (scripting language) | Graphical (UML activities like notation) | Textual (QVT-O) | Textual (RubyTL) |
| Cross-language-support | Yes/no | No (ATL) | Yes (ATL, Java, MTF) | Yes | Yes | No (ATL), but extension point for calling generators | No (QVT-O) | No (RubyTL) |
| Persist intermediate results | ✓ (Supported)/ × (Not supported)/ ? (Unknown) | ✓ | ✓ (All results persisted automatically) | ? | ? | ✓ (Must be explicitly modeled by artifacts) | ✓ (Must be explicitly specified by `out` parameters) | ✓ |
| Composition expressivity | | | | | | | | |
| Orchestration as black-box (nesting) | ✓ (Supported)/ × (Not supported)/ ? (Unknown) | ✓ | ✓ | ✓ | × | × | × | ✓ |
| Sequential composition | ✓ (Supported)/ × (Not supported)/ ? (Unknown) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Conditional composition | ✓ (Supported)/ × (Not supported)/ ? (Unknown) | ✓ | × | × | ✓ | ✓ | ✓ | ✓ |
| Repetitive composition | ✓ (Supported)/ × (Not Supported)/ (Unknown) | ✓ | × | × | × | × | ✓ | ✓ |
| Parallel composition | ✓ (Supported)/ × (Not supported)/ ? (Unknown) | ✓ | ✓ | ✓ | ✓ | × | × | × |
| *Static semantics* | | | | | | | | |
| Model validation | [Compile-time/ run-time/no] error | Run-time error | Run-time error | Run-time error | Run-time error | Run-time error | Run-time error | Run-time error |
| Type checking | [Compile-time/ run-time/no] error | Run-time error | Compile-time error | Run-time error | Compile-time error | Run-time error | Run-time error | Run-time error |
| *Dynamic semantics* | | | | | | | | |
| Execution order | Specification-order/undefined | Undefined | Specification-order | Undefined | Undefined | Undefined | Specification-order | Specification-order |
| Cross-transformation traceability | Yes/no | No | No | No | No | No | No | No |

models, i.e., *model validation* should take place. Second, it must be ensured that actual parameters of the transformation orchestration are valid types given the formal parameters of the transformation orchestration, i.e., *type checking* should take place.

Regarding *dynamic semantics*, it is first examined, how the transformation orchestration is executed, i.e., the *execution order*, which is of special interest in case that parallel execution of several model transformations is possible. Second, it is relevant whether a transformation execution is able to establish *cross-transformation traceability*, i.e., whether a given source model element might be traced across the whole transformation chain to the resulting model element(s).

**Evaluation of languages** Concerning the *syntax of orchestration mechanisms*, all the languages evaluated except QVT-O and RubyTL are *extrinsic* to a specific transformation language. The employed *formalisms* to specify orchestrations of model transformations are in four cases *graphical*, whereby only MCC, QVT-O, and RubyTL rely on a *textual* specification. With respect to *cross-language support*, UniTI, the Transformation Composition Modeling Framework, and MCC allow for the orchestration of transformations written in different transformation languages. In contrast, Wires*, ATLflow, QVT-O, and RubyTL are specific to a single transformation language, only. Regarding the possibility to *persist intermediate results* of a transformation chain, all languages seem to provide support, whereby in two cases, this property is unknown, since the corresponding tools could not be tested, and thus, a paper-based evaluation has taken place. The *expressivity* of the different orchestration languages mainly differs in the support for *nested orchestrations*, which are allowed by Wires*, UniTI, the Transformation Composition Modeling Framework, and RubyTL, only and in the support for *repetitive compositions*, which are allowed by Wires*, QVT-O, and RubyTL.

With respect to *static semantics*, one may see that checking thereof is still limited in the various orchestration languages. Instead, run-time errors occur concerning *model validation* and *type checking*.

Finally, regarding *dynamic semantics* of orchestration languages, most of the languages do not expose the internal *execution order* to the transformation designer, e.g., in case of parallel execution. Furthermore, the support for *cross-transformation traceability* is not available in a single orchestration language.

#### *3.6.2 Synopsis*

Orchestration is a promising approach for reusing transformation logic without specialization efforts. Nevertheless, the frequency of occurrence is constrained by the specificity of the reused transformations, since each one is bound to concrete source and target metamodels. Thus, it might be beneficial to combine orchestration with generic transformations [84]. Furthermore, to allow specialization of the chained transformations, module import as well as HOTs, AOP, and reflection seems to be supplementary for transformation chains.

### 3.7 Summary of comparison of reuse mechanisms

Table 12 gives an overview on the comparison results. Thereby, a language may either provide *no support* (indicated by a cross), *direct support* (indicated by a check mark), or *indirect support*, i.e., additional reuse approaches are defined on top of the language (indicated by a check mark in parentheses). From this summary, one may conclude that some languages already provide some support for all reuse scenarios (such as Kermeta and ATL), although not all reuse mechanisms are supported. The most supported reuse scenario is scenario 1, which is also considered to be the simplest scenario, since being the scenario of concrete intra-transformation reuse in the small. In contrast, the reuse scenarios 2–5 are scarcely supported, which is further detailed in the following section, discussing current barriers and facilitators to model transformation reuse. However, it has to be noted that for scenario 2 and for scenario 3, modules and HOTs are the current mechanisms of choice, respectively.

## 4 Barriers and facilitators to model transformation reuse

As may be seen in Sect. 3 and in the summarizing Table 13, numerous reuse mechanisms have been proposed in literature. Nevertheless, when examining the results in Table 13, severe barriers to model transformation reuse may be deduced, hindering the successful adoption in practice. In the following, the main barriers derived from our comparison are presented, identifying further research potentials. Additionally, first facilitators for model transformation reuse are given.

**Insufficient abstraction from metamodels** As may be seen in Table 13, many of the proposed reuse mechanisms depend on concrete metamodel types. Additionally, those allowing to decouple transformation logic from concrete metamodel types, as, e.g., genericity, are nevertheless still dependent on the internal structure of the metamodels. Thus, reuse of transformation logic between different metamodels is hampered. To improve this situation, not only standardized metamodels but also standardized transformations might be beneficial. By extending the standardized metamodels, also the standardized transformations might be reused to realize specific transformations, resembling the idea of *subtyping* for model transformations [71]. Based on this, hierarchies of

**Table 12** Overview on reuse support of investigated languages

| Reuse scenario | Imperative | | Declarative | | | | Hybrid | | |
|---|---|---|---|---|---|---|---|---|---|
| | Kermeta | QVT-O | QVT-R | TGGs | Viatra | TNs | ATL | ETL | RubyTL |
| *Reuse scenario 1* | | | | | | | | | |
| Code scavenging | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| User-defined functions | ✓ | ✓ | ✓ | × | ✓ | × | ✓ | ✓ | ✓ |
| Rule inheritance | ✓ | ✓ | × | ✓ | × | ✓ | ✓ | ✓ | × |
| *Reuse scenario 2* | | | | | | | | | |
| Module import | ✓ | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ |
| Transformation product lines | × | × | × | × | × | × | (✓) | × | × |
| *Reuse scenario 3* | | | | | | | | | |
| HOT | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| AOP | ✓ | × | × | × | × | × | × | × | × |
| Reflection | ✓ | × | × | ✓ | × | × | (✓) | ✓ | ✓ |
| Generic functions | ✓ | × | × | ✓ | ✓ | × | × | × | × |
| Embedded DSLs | × | × | × | × | × | × | (✓) | × | × |
| *Reuse scenario 4* | | | | | | | | | |
| Generic transformations | ✓ | × | × | × | ✓ | × | (✓) | × | × |
| Stand-alone DSLs | × | × | × | × | × | (✓) | (✓) | × | × |
| *Reuse scenario 5* | | | | | | | | | |
| Orchestration | ✓ | ✓ | × | × | × | × | (✓) | × | ✓ |

model transformations may be defined between hierarchies of metamodels to enable transformation reuse in practice. Nevertheless, additional costs for connecting standardized metamodels to specific metamodels may occur, because of structural heterogeneities between them [35].

**Insufficient abstraction from transformation languages** Except stand-alone DSLs, all reuse mechanisms target a single transformation language as may be seen in Table 13. Consequently, there is little work on how to reuse transformation logic across transformation language boundaries. A first step in this direction is presented in [81], where a classification of structural heterogeneities in model-to-model transformations is given, which may serve as a pattern library for model transformations. Furthermore, reusable transformation patterns have been presented in [1] for graph transformation languages and idioms for QVT in [36]. Going one step further, reuse should be enabled during the whole development cycle including also requirements analysis, design, and testing to guarantee the development of high-quality model transformations [32,65].

**Missing repositories for selection** As may be seen from our comparison and in Table 13, hardly any repository of reusable artifacts has been established so far, except the ATL model transformation zoo. However, the zoo basically constitutes a collection of ATL transformations and was thus not explicitly designed for reuse. This is similar to research on transformation libraries, since the main focus was on the standard OCL libraries and their extensions [12,15]. However, a general mechanism to provide user-defined OCL library extensions in a flexible and systematic manner is missing [6]. This undesirable situation is in contrast to software engineering, where different kinds of repositories of reusable artifacts exist, ranging from fine-grained class libraries over components to coarse-grained frameworks which can be reused out of the box. In this respect, repositories with means to efficiently select reusable artifacts are key to successful reuse in model transformations.

**Lack of meta-information in selection** Apart from missing repositories for selection, there is also little meta-information available for selecting a reusable artifact without having to know its internals, as Table 13 reveals. Therefore, it would be important to provide transformations with according meta-information, comprising source/target metamodels, test models, pre- and postconditions, and documentation. Preconditions may be used, e.g., to check whether input models conform to the implemented transformation logic [13]. More abstract models for model transformations, e.g., requirements, would provide an additional source of meta-information. To support the transformation designer in finding reusable artifacts, transformation search engines, in analogy to model search engines such as Moogle [50], seem to be an interesting future research direction. Such search engines may base on recent advances in metamodel matching [26,38,77] as a starting point. Furthermore, model

**Table 13** Conceptual comparison of reuse mechanisms

| | Scenario 1: concrete intra-transformation reuse in the small | | | Scenario 2: concrete intra-transformation reuse in the large | | Scenario 3: generic inter-transformation reuse in the small | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Code scavenging | User-defined functions | Rule inheritance | Module import | Transformation product lines | HOT | AO | Reflection | Generic functions | Embedded DSLs |
| Reusable artifact | Code part | Function | Base rules | Base transformation | Feature model, transformation | Base transformation, or HOT | Advice | Reflective transformation | Generic rules | DSL constructs, generator |
| *Abstraction* | | | | | | | | | | |
| Generalization | | | | | | | | | | |
| From MM | Manual | × | × | × | × | ✓ | ✓ | ✓ | ✓ | ✓ |
| From TL | × | × | × | × | × | × | × | × | × | × |
| Simplification | | | | | | | | | | |
| Hidden parts | None | Implementation | None | None | Transformation | None | None | None | Implementation | Operational semantics |
| Visible parts | All | Signature | All | All | Features | All | All | All | Signature with type parameters | DSL syntax |
| *Selection* | | | | | | | | | | |
| Repository | Own transformation | Own libraries | Own transformation | Transformation repository (e.g. ATL zoo) | × | Transformation repository (e.g. ATL zoo) | × | × | × | DSL constructs |
| Metainfo | – | – | – | Documentation | – | Documentation | – | – | – | Documentation, Grammar |
| Automatism | Manual | Manual | Manual | Manual | – | Manual | – | – | – | Semi-automatic (code completion) |
| *Specialization* | | | | | | | | | | |
| Required knowledge | White-box | Black-box | White-box | Black-box | Black-box | Black-box | Black-box | Black-box | Black-box | Black-box |
| Mechanism | – | Parameter binding | Rule overriding | Rule redefintion | Configuration on basis of the feature model | Transformation | Join point model | Metarules | Type parameter binding | Parameter binding |
| Language-inherent | ✓ | ✓ | ✓ | ✓ | × | ✓ | ✓ | ✓ | ✓ | ✓ |
| *Integration* | | | | | | | | | | |
| Ability | Composition | Composition | Composition | Composition | Generation | Composition | Composition | Composition | Composition | Generation |
| Kind | Containment | Connection | Extension | Extension | – | Extension | Extension | Extension | Connection | – |

**Table 13** continued

| | Scenario 4: generic inter-transformation reuse in the large | | Scenario 5:concrete inter-transformation reuse in the large |
|---|---|---|---|
| | Generic transformations | Stand-alone DSLs | Orchestration |
| Reusable artifact | Generic transformations | DSL constructs, generator | Transformation |
| *Abstraction* | | | |
| Generalization | | | |
| From MM | ✓ | ✓ | × |
| From TL | × | ✓ | × |
| Simplification | | | |
| Hidden parts | Implementation | Operational semantics | Implementation |
| Visible parts | Signature with type parameters | DSL syntax | Signature (source/target MMs) |
| *Selection* | | | |
| Repository | × | DSL constructs | Transformation repository (e.g. ATL zoo) |
| Metainfo | – | Documentation, Grammar | Documentation |
| Automatism | – | Semi-automatic (code completion) | Manual |
| *Specialization* | | | |
| Required knowledge | Black-box | Black-box | – |
| Mechanism | Type parameter binding | Parameter binding | – |
| Language-inherent | × (Extrinsic binding) | ✓ | – |
| | ✓ (Intrinsic binding) | | |
| *Integration* | | | |
| Ability | Generation (extrinsic) | Generation | Composition |
| | Composition (intrinsic) | | |
| Kind | Connection (intrinsic) | – | Coordination |

transformation testing is currently gaining a huge momentum in the research landscape of model transformations. Several approaches for (i) defining contracts as specifications for transformations [29,33] and (ii) (semi)-automatic generation of test input models [30,31,61] are emerging. Going further in all these mentioned directions will contribute to the quality of model transformations, and in return, to their reuse.

**Challenging specialization** As may be seen in Table 13, most reuse mechanisms allow for specialization. However, the specialization is often challenging to be applied in practice. This includes especially HOTs as also stated by Tisi et al. [67], where the user must be familiar with the abstract syntax of the transformation language. To speed up development, Kühne et al. [45] proposed to work with the concrete syntax of the transformation language instead. The situation is even worse in case of rule inheritance, since specialization is limited. This may be derived from the fact that none of the approaches allows to define reuse policies, e.g., to disallow rule inheritance (cf. `final` keyword in Java) or to define some access rights (cf. keywords `private`, `protected` or `public`) [83].

**Insufficient support for integration in the large** Although orchestration languages have been proposed to chain transformations (cf. Table 11), a main issue is the compatibility of source/target metamodels between the orchestrated transformations [75]. Thus, mechanisms are needed to ensure type compatibility in transformation chains similar to type checks in ordinary programs. This would incorporate compilation errors, if compatibility between metamodels is violated. However, there may be additional conditions for meaningful chaining of model transformations [14]. For example, if the former transformation in a transformation chain produces a UML class diagram that conforms to the UML metamodel, and the subsequent transformation expects a UML activity diagram which also conforms to the UML metamodel, the latter one will never be able to produce any result, although typing is correct. Emerging work on estimating footprints of model transformations on the metamodel [11,37] may serve as foundation to reason about compatibility of transformations.

**Reuse in practice: are we there yet?** Finally, the question raised by the title of the paper should be answered. For this, we conducted a case study[11] with content of the ATL model transformation zoo. In this context, we analyzed, which reuse mechanisms have been employed in designing the transformations. Results of this case study show that around 80 % of the transformations available in the zoo make use of user-defined functions. Furthermore, 4 % represent HOTs and may thus be reused themselves and 11 % are orchestrations of model transformations. Interestingly, although around 75 % of the metamodels available in the zoo make use of inheritance, only 4 % of the transformations apply rule inheritance, and thus, potential for reuse is wasted. Finally, none of the transformations in the zoo makes use of superimposition and TPLs. These numbers show that reuse for scope 1 is performed in practice. In contrast, reuse across transformation boundaries and metamodel boundaries is still in its infancy. Consequently, the question whether reuse has found its way to practice may be answered in the negative.

[11] Online at http://www.modeltransformation.net.

## 5 Conclusion

Model transformation reuse has made major advances in the last years. A huge number of papers have been published on this topic, and model transformation languages offer several reuse mechanisms out of the box or additionally built on top of them. In this paper, we have provided an overview on the proposed reuse mechanisms for M2M transformations. First, we have conducted a conceptual comparison of reuse mechanisms on the basis of a framework covering (i) the main dimensions of transformation reuse mechanisms to cluster reuse scenarios and (ii) the main phases in the reuse process, comprising abstraction, selection, specialization, and integration. Second, we compared the realization of the reuse mechanisms in different M2M transformation languages. Based on this comparison, we conclude that a variety of mechanisms for reuse are available in current model transformation languages; however, their concrete realization may differ between the different languages. Third, we identified several barriers during our studies hindering the successful and broad application of reuse mechanisms, but also facilitators that may help to overcome the barriers in the future.

Finally, we want to emphasize that currently there is a strong focus on reuse in the implementation phase, but reuse across all development phases would be urgently needed, e.g., general guidelines on how to design transformations. Thus, in our opinion, further research is needed to anchor model transformation reuse as a central part of model transformation engineering.

**Acknowledgments** This work has been funded by the Austrian Science Fund (FWF) under grant P21374-N13 and J3159-N23 as well as by the Austrian Research Promotion Agency (FFG) under project number 832160.

## References

1. Agrawal, A., Vizhanyo, A., Kalmar, Z., Shi, F., Narayanan, A., Karsai, G.: Reusable idioms and patterns in graph transformation

languages. Electron. Notes Theor. Comput. Sci. **127**(1), 181–192 (2005)
2. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA (1986)
3. Allen, E., Cartwright, R., Stoler, B.: Efficient implementation of run-time generic types for Java. In: Proceedings of the IFIP Working Conference on Generic Programming, vol. 243 of IFIP Conference Proceedings, pp. 207–236. Kluwer (2002)
4. Amrani, M., Dingel, J., Lambers, L., Lucio, L., Salay, R., Selim, G., Syriani, E., Wimmer, M.: Towards a model transformation intent catalog. In: Proceedings of the 1st International Workshop on the Analysis of Model Transformations (AMT) @ MoDELS'12, pp. 3–8. ACM (2012)
5. AtlanMod Team. ATL website. http://www.eclipse.org/atl. Last accessed: March 2013
6. Baar, T.: On the need of user-defined libraries in OCL. ECEASST, 36 (2010)
7. Balasubramanian, D., Narayanan, A., van Buskirk, C.P., Karsai, G.: The graph rewriting and transformation language: GReAT. ECEASST 1 (2006)
8. Bézivin, J.: On the unification power of models. Softw. Syst. Model. **4**(2), 171–188 (2005)
9. Bézivin, J., Rumpe, B., Schürr, A., Tratt, L.: Model transformations in practice workshop @ MoDELS'05. Montego Bay, Jamaica (2005)
10. Biggerstaff, T.J., Richter, C.: Reusability framework, assessment, and diretions. IEEE Softw. **4**(2), 41–49 (1987)
11. Burgueno, L., Wimmer, M., Vallecillo, A.: Towards tracking guilty transformation rules. In Proceedings of the 1st International Workshop on the Analysis of Model Transformations @ MoDELS'12 (2012)
12. Cabot, J., Mazón, J.-N., Pardillo, J., Trujillo, J.: Specifying aggregation functions in multidimensional models with OCL. In: Proceedings of 29th International Conference on Conceptual Modeling (ER'10), vol. 6412 of Lecture Notes in Computer Science, pp. 419–432. Springer (2010)
13. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. ECEASST, 24 (2009)
14. Chenouard, R., Jouault, F.: Automatically discovering hidden transformation chaining constraints. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), vol. 5795 of Lecture Notes in Computer Science, pp. 92–106. Springer (2009)
15. Chimiak-Opoka, J.: OCLLib, OCLUnit, OCLDoc: pragmatic extensions for the object constraint language. In: Proceedings of the 12th international conference on model driven engineering languages and systems (MODELS'09), vol. 5795 of lecture notes in computer science, pp. 665–669. Springer (2009)
16. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, Reading, MA (2001)
17. Cuadrado, J., García Molina, J.: Approaches for model transformation reuse: factorization and composition. In: Proceedings of the 1st International Conference on Theory and Practice of Model Transformations (ICMT'08), vol. 5063 of Lecture Notes in Computer Science, pp. 168–182. Springer (2008)
18. Cuadrado, J., Guerra, E., de Lara, J.: Generic model transformations: write once, reuse everywhere. In: Proceedings of the 4th International Conference on Theory and Practice of Model Transformations (ICMT'11), vol. 6707 of Lecture Notes in Computer Science, pp. 62–77. Springer (2011)
19. Cuadrado, J., Jouault, F., García Molina, J., Bézivin, J.: Experiments with a high-level navigation language. In: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'09), vol. 5563 of Lecture Notes in Computer Science, pp. 229–238. Springer (2009)
20. Cuadrado, J., Molina, J.G.: A model-based approach to families of embedded domain-specific languages. IEEE Trans. Softw. Eng. **35**, 825–840 (2009)
21. Cuadrado, J.S., Molina, J.G.: Modularization of model transformations through a phasing mechanism. Softw. Syst. Model. **8**(3), 325–345 (2009)
22. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3), 621–645 (2006)
23. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Proceedings of the 3rd international conference on software product lines (SPLC'04), vol. 3154 of Lecture Notes in Computer Science, pp. 266–283. Springer (2004)
24. Del Fabro, M., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Softw. Syst. Model. **8**(3), 305–324 (2009)
25. Eclipse Community. QVT-O website. http://www.eclipse.org/mmt/?project=qvto. Last accessed: March 2013
26. Falleri, J.-R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'08), vol. 5301 of Lecture Notes in Computer Science, pp. 326–340. Springer (2008)
27. France, R., Rumpe, B.: Model-driven development of complex software: a research roadmap. In: Proceedings of the Workshop on the Future of Software Engineering @ ICSE'07, pp. 37–54. IEEE Computer Society (2007)
28. George, L., Wider, A., Scheidgen, M.: Type-safe model transformation languages as internal DSLs in scala. In: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT'12), vol. 7307 of Lecture Notes in Computer Science, pp. 160–175. Springer (2012)
29. Gogolla, M., Vallecillo, A.: Tractable model transformation testing. In: Proceedings of the 7nd European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA'11), vol. 6698 of Lecture Notes in Computer Science, pp. 221–235. Springer (2011)
30. González, C.A., Cabot, J.: ATLTest: a white-box test generation approach for ATL transformations. In: Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems (MODELS'12), vol. 7590 of Lecture Notes in Computer Science, pp. 449–464. Springer (2012)
31. Guerra, E.: Specification-driven test generation for model transformations. In: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT'12), vol. 7307 of Lecture Notes in Computer Science, pp. 40–55. Springer (2012)
32. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: transML: a family of languages to model model transformations. In: Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10), vol. 6394 of Lecture Notes in Computer Science, pp. 106–120. Springer (2010)
33. Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. J. Autom. Softw. Eng. **20**(1), 5–46 (2013)
34. Günther, S., Cleenewerck, T.: Design principles for internal domain-specific languages: a pattern catalog illustrated by Ruby. In: Proceedings of the 17th International Conference on Pattern Languages of Programs (PLoP'10) (2010)
35. Guy, C., Combemale, B., Derrien, S., Steel, J., Jézéquel, J.-M.: On model subtyping. In: Proceedings of the 8th European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA'12), vol. 7349 of Lecture Notes in Computer Science, pp. 400–415. Springer (2012)

36. Iacob, M.-E., Steen, M.W.A., Heerink, L.: Reusable model transformation patterns. In: Workshop Proceedings of the 12th Enterprise Distributed Object Computing Conference (EDOCW'08), pp. 1–10. IEEE Computer Society (2008)
37. Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE'11), pp. 601–610. ACM (2011)
38. Kappel, G., Kargl, H., Kramler, G., Schauerhuber, A., Seidl, M., Strommer, M., Wimmer, M.: Matching metamodels with semantic systems—an experience report. In: Workshop Band der Fachtagung Datenbanksysteme in Business, Technologie und Web (BTW'07), pp. 38–52 (2007)
39. Kavimandan, A., Gokhale, A., Karsai, G., Gray, J.: Templatized model transformations: enabling reuse in model transformations. Vanderbilt University, Technical report (2009)
40. Kiczales, G.: Aspect-oriented programming. ACM Comput. Surv. **28**(4), 154 (1996)
41. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE'07), pp. 285–294. ACM (2007)
42. Kleppe, A.: MCC: a model transformation environment. In: Proceedings of the 2nd European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA'06), vol. 4066 of Lecture Notes in Computer Science, pp. 173–187. Springer (2006)
43. Kramer, J.: Is abstraction the key to computing? Commun. ACM **50**, 36–42 (2007)
44. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2), 131–183 (1992)
45. Kühne, T., Mezei, G., Syriani, E., Vangheluwe, H., Wimmer, M.: Explicit transformation modeling. In: Models in Software Engineering-Reports and Revised Selected Papers of Workshops and Symposia at MoDELS'09, vol. 6002 of Lecture Notes in Computer Science, pp. 240–255. Springer (2009)
46. Kurtev, I.: Application of reflection in a model transformation language. Softw. Syst. Model. **9**(3), 311–333 (2010)
47. Lau, K., Rana, T.: A taxonomy of software composition mechanisms. In: Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'10), pp. 102–110. IEEE (2010)
48. Legros, E., Amelunxen, C., Klar, F., Schürr, A.: Generic and reflective graph transformations for checking and enforcement of modeling guidelines. Vis. Lang. Comput. **20**(4), 252–268 (2009)
49. Liskov, B., Wing, J.M.: A new definition of the subtype relation. In: Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93), vol. 707 of Lecture Notes in Computer Science, pp. 118–141. Springer (1993)
50. Lucrédio, D., de Mattos Fortes, R.P., Whittle, J.: MOOGLE: a metamodel-based model search engine. Softw. Syst. Model. **11**(2), 183–208 (2012)
51. Ma, H., Shao, W., Zhang, L., Ma, Z., Jiang, Y.: Applying OO metrics to assess UML meta-models. In: Proceedings of the 7th international conference on the unified modelling language (UML'04), vol. 3273 of Lecture Notes in Computer Science. Springer (2004)
52. Mili, A., Mili, R., Mittermeir, R.: A survey of software reuse libraries. Ann. Softw. Eng. **5**, 349–414 (1998)
53. Mili, H., Mili, F., Mili, A.: Reusing software: issues and research directions. IEEE Trans. Softw. Eng. **21**(6), 528–562 (1995)
54. Moha, N., Mahé, V., Barais, O., Jézéquel, J.-M.: Generic model refactorings. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), vol. 5795 of Lecture Notes in Computer Science, pp. 628–643. Springer (2009)
55. Oldevik, J.: Transformation composition modelling framework. In: Proceedings of the 5th international conference on distributed applications and interoperable systems (DAIS'05), vol. 3543 of Lecture Notes in Computer Science, pp. 108–114. Springer (2005)
56. Randak, A., Martínez, S., Wimmer, M.: Extending ATL for native UML profile support: an experience report. In: Proceedings of the 3rd International Workshop on Model Transformation with ATL (MtATL'11), pp. 49–62. CEUR Workshop Proceedings (2011)
57. Reiter, T., Wimmer, M., Kargl, H.: Towards a runtime model based on colored petri-nets for the execution of model transformations. In: Proceedings of the 3rd International Workshop on Models and Aspects @ ECOOP'07, pp. 19–23 (2007)
58. Rivera, J.E., Ruiz-Gonzalez, D., Lopez-Romero, F., Bautista, J., Vallecillo, A.: Orchestrating ATL model transformations. In: Proceedings of the 1st International Workshop on Model Transformations with ATL (MtATL'09), pp. 34–46 (2009)
59. Schmidt, D.C.: Model-driven engineering. IEEE Comput. **39**(2), 25–31 (2006)
60. Schönböck, J.: Testing and debugging of model transformations. Ph.D. thesis, Vienna University of Technology, Business Informatics Group (2011)
61. Sen, S., Mottu, J.-M., Tisi, M., Cabot, J.: Using models of partial knowledge to test model transformations. In: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT'12), vol. 7307 of Lecture Notes in Computer Science, pp. 24–39. Springer (2012)
62. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Softw. **20**(5), 42–45 (2003)
63. Sijtema, M.: Introducing variability rules in ATL for managing variability in MDE-based product lines. In: Proceedings of the 2nd International Workshop on Model Transformations with ATL (MtATL'10), pp. 39–49. CEUR Workshop Proceedings (2010)
64. Steel, J., Jézéquel, J.-M.: On model typing. Softw. Syst. Model. **6**, 401–413 (2007)
65. Syriani, E., Gray, J.: Challenges for addressing quality factors in model transformation. In: Proceedings of the 5th International Conference on Software Testing, Verification and Validation (ICST'12), pp. 929–937. IEEE (2012)
66. TATA Research Development and Design. ModelMorf website. http://www.tcs-trddc.com/trddc_website/ModelMorf/ModelMorf.htm. Last accessed: March 2013
67. Tisi, M., Cabot, J., Jouault, F.: Improving higher-order transformations support in ATL. In: Proceedings of the 3th International Conference on Theory and Practice of Model Transformations (ICMT 2010), vol. 6142 of Lecture Notes in Computer Science, pp. 215–229. Springer (2010)
68. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Proceedings of the 5th European Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA'09), vol. 5562 of Lecture Notes in Computer Science, pp. 18–33. Springer (2009)
69. Triskell Team. Kermeta website. http://www.kermeta.org. Last accessed: March 2013
70. University of York. ETL Website. http://www.eclipse.org/epsilon/doc/etl. Last accessed: March 2013
71. Vallecillo, A., Gogolla, M.: Typing model transformations using tracts. In: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT'12), vol. 7307 of Lecture Notes in Computer Science, pp. 56–71. Springer (2012)
72. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers Y.: UniTI: a unified transformation infrastructure. In: Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS'07), vol. 4735 of Lecture Notes in Computer Science, pp. 31–45. Springer (2007)

73. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3), 214–234 (2007)
74. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: Proceedings of the 7th International Conference on the Unified Modelling Language (UML'04), vol. 3273 of Lecture Notes in Computer Science, pp. 290–304. Springer (2004)
75. Vignaga, A., Jouault, F., Bastarrica, M.C., Bruneliere, H.: Typing in model management. In: Proceedings of the 2th International Conference on Theory and Practice of Model Transformations (ICMT'09), vol. 5563 of Lecture Notes in Computer Science, pp. 197–212. Springer (2009)
76. Voelter, M., Groher, I.: Handling variability in model transformations and generators. In: Proceedings of the 7th International Workshop on Domain-Specific Modeling @ OOPSLA '07 (2007)
77. Voigt, K., Heinze, T.: Metamodel matching based on planar graph edit distance. In: Proceedings of the 3th International Conference on Theory and Practice of Model Transformations (ICMT'10), vol. 6142 of Lecture Notes in Computer Science, pp. 245–259. Springer (2010)
78. Wagelaar, D.: Composition techniques for rule-based model transformation languages. In: Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, 1–2 July 2008, Proceedings, vol. 5063 of Lecture Notes in Computer Science, pp. 152–167. Springer (2008)
79. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Softw. Syst. Model. **9**, 285–309 (2010)
80. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Surviving the heterogeneity jungle with composite mapping operators. In: Proceedings of the 3nd International Conference on Theory and Practice of Model Transformations (ICMT'10), vol. 6627 of Lecture Notes in Computer Science, pp. 260–275. Springer (2010)
81. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Towards an expressivity benchmark for mappings based on a systematic classification of heterogeneities. In: Proceedings of the International Workshop on Model-Driven Interoperability @ MoDELS 2010, pp. 32–41 (2010)
82. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Fact or fiction—reuse in rule-based model-to-model transformation languages. In: Proceedings of the 5th International Conference on Theory and Practice of Model Transformations (ICMT'12), vol. 7307 of Lecture Notes in Computer Science, pp. 280–295. Springer (2012)
83. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D., Paige, R., Lauder, M., Schürr, A., Wagelaar, D.: Surveying rule inheritance in model-to-model transformation languages. J. Object Technol. **11**(2), 3:1–46 (2012)
84. Wimmer, M., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Sánchez Cuadrado, J., Guerra, E., De Lara, J.: Reusing model transformations across heterogeneous metamodels. In: Proceedings of the 5th International Workshop on Multi-Paradigm Modeling @ MoDELS'11 (2011)
85. Wimmer, M., Schauerhuber, A., Kappel, G., Retschitzegger, W., Schwinger, W., Kapsammer, E.: A survey on UML-based aspect-oriented design modeling. ACM Comput. Surv. **43**(4), 28 (2011)

## Author Biographies

**A. Kusel** is a postdoctoral researcher at the Cooperative Information Systems Group at the Johannes Kepler University, Linz. Her current research interests include model engineering, in particular the specification of model transformations. She received a PhD in Computer Science from the Johannes Kepler University in 2011 in the area of model engineering entitled "Reusability in Model Transformations—Resolving Recurring Heterogeneities by Composite Mapping Operators." For further information about her research activities, please visit http://www.tk.jku.at/people or contact her at e-mail: Angelika.Kusel@jku.at.

**J. Schönböck** is a professor at the Upper Austrian University of Applied Sciences in Hagenberg. He received a PhD in Computer Science from the Vienna University of Technology in 2012 for his thesis "Testing and Debugging of Model Transformations." His research comprise model transformations, model transformation testing and debugging, model (co-)evolution. For further information about his research activities, please visit http://research.fh-ooe.at/de/staff/2113 or contact him at e-mail: johannes.schoenboeck@fh-hagenberg.at.

**M. Wimmer** is post-doc researcher at the Business Informatics Group of the Vienna University of Technology. His research interests comprise Web engineering and model engineering, in particular model transformations based on formal methods, generating transformations by example as well as applying model transformations to deal with model evolution. During the time of writing of this article, he has been working as visiting researcher at the Software Engineering Group of the University of Málaga (Spain). For further information about his research activities, please visit http://www.big.tuwien.ac.at/staff/mwimmer or contact him at e-mail: wimmer@big.tuwien.ac.at.

**G. Kappel** is a full professor in the Institute for Software Technology and Interactive Systems at the Vienna University of Technology, heading the Business Informatics Group. Her current research interests include model engineering, Web engineering, as well as process engineering. For further information about her research activities, please contact her at e-mail: gerti@big.tuwien.ac.at or visit http://www.big.tuwien.ac.at/staff/gkappel.

**W. Retschitzegger** is associate professor at the Johannes Kepler University (JKU) Linz, Austria, and scientific head of the Department of Cooperative Information Systems (CIS). In 2002, he was appointed a temporary full professorship for business informatics at TU Vienna. From 2003 to 2006, he was chair of the Institute of Bioinformatics at JKU; from 2008 to 2009, he held a guest professorship for workflow management at the University of Vienna. He has published more than 150 papers in international refereed journals and conference proceedings, along with a series of books. His research interests focus on the model-driven and semantic-based engineering of cooperative information systems in domains like road traffic management, workflow management, and social media. Contact him at e-mail: werner@ifs.uni-linz.ac.at, or visit http://www.bioinf.jku.at/people/wr/.

**W. Schwinger** is associate professor at the Department of Cooperative Information Systems (CIS) at the Johannes Kepler University (JKU) Linz, Austria, and jointly is acting as scientific head of department. Prior to that, he was working as a senior researcher and project manager of strategic research projects at the Software Competence Center Hagenberg, Austria. He was involved in several national and international projects in the areas of context and situation aware systems, model engineering, and Web engineering resulting in more than 90 publications in international refereed journals and conference proceedings. Contact him at e-mail: wieland@schwinger.at, or visit http://www.tk.uni-linz.ac.at/people/.