Fact or Fiction – Reuse in Model-to-Model Transformations*

M. Wimmer[1], G. Kappel[1], A. Kusel[2],
W. Retschitzegger[2], J. Schönböck[1], and W. Schwinger[2]

[1] Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
{firstname.lastname}@jku.at

Abstract. Model transformations are mostly developed from scratch. 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. Therefore, this paper provides an in-depth comparison of reuse mechanisms in rule-base model-to-model transformation languages and categorizes them along their intended scope of application. For this, a systematic comparison framework for reuse mechanisms is proposed to highlight commonalities as well as differences. Finally, current barriers to model transformation reuse are outlined.

Key words: Reuse Mechanisms, Model Transformations, Comparison

1 Introduction

Model transformations are crucial for the success of Model-Driven Engineering, being comparable in role and importance to compilers for high-level programming languages. Nevertheless, most of today's transformation designers still follow an ad-hoc manner to specify model transformations [10]. For increasing development productivity as well as quality of model transformations, the application of appropriate reuse mechanisms is indispensable. This need has been recognized by the research community as a plethora of proposed reuse mechanisms reveals [6–9, 13, 16, 18, 21–23, 25–29]. Nevertheless, there exists no survey providing an overview of the proposed mechanisms to deeper understand their commonalities and differences. Thus, it is unclear which reuse mechanisms may be employed in a certain situation and which barriers exist in applying them.

Therefore, this paper provides an in-depth comparison of proposed reuse mechanisms in rule-based model-to-model transformation languages to highlight when to apply a certain reuse mechanism and how reuse mechanisms complement each other. In this respect, reuse mechanisms are categorized along their intended scope of application, ranging from *reuse in the small*, e.g., functions, to *reuse*

* This work has been funded by the FWF under grant P21374-N13.

in the large, e.g., orchestration of model transformations. For this, a systematic comparison framework for reuse mechanisms is proposed comprising comparison criteria along four different dimensions analogous to the main phases in software reuse [15], being *abstraction*, *selection*, *specialization* and *integration*. On the basis of this framework, the categorized reuse mechanisms are compared and for each reuse mechanism corresponding supporting representatives are given. To illustrate the different mechanisms, example reuse scenarios on the basis of a running example are given.

Outline. Section 2 introduces the running example and presents the comparison framework with its four dimensions. In Section 3, the comparison framework is used to compare the reuse mechanisms along their different scopes of reuse. Section 4 presents barriers to reuse in model transformations and finally, Section 5 concludes the paper.

2 Comparison Framework

This section introduces scopes of reuse based on an example which is used throughout the paper to illustrate the different reuse mechanisms. Furthermore, a comparison framework is presented to characterize the reuse mechanisms.

2.1 Scopes of Reuse

Different scopes of reuse exist which posses different reuse potentials, e.g., within/across transformations or between the same/different metamodels (MMs). In this respect, we identified five different scopes ranging from *reuse in the small* to *reuse in the large* which are depicted in Fig. 1 on basis of the `Class2ER` example [2]:

- *Scope 1:* To avoid code duplication, reuse of logic within *a single transformation* is needed, i.e., the scope is to reuse the same transformation logic between the *same MMs* in the *same transformation* (cf. (1) in Fig. 1).
- *Scope 2:* To realize similar transformation logic, e.g., to pursue different OR-mapping approaches – a "one table per hierarchy" approach instead of a "one table per class" approach (cf. (2) in Fig. 1) – reuse of transformation logic between the *same MMs* in *different transformations* is needed.
- *Scope 3:* The transformation logic of the `Class2ER` example might be needed in an `Ontology2XML` transformation as well, requiring that the same transformation logic could be reused in the context of *different MMs* and thus *different transformations* (cf. (3) in Fig. 1).
- *Scope 4:* Since cross-cutting concerns, e.g., debugging or tracing (cf. (4) in Fig. 1), should be reusable throughout transformations, mechanisms are needed that allow to reuse *logic* irrespective of *MMs* and *transformations*.
- *Scope 5:* Reuse in the large is achieved when existing transformations can be applied *without changing transformation logic or MMs*, as is the case for chaining a `ER2Relational` transformation after the `Class2ER` transformation in our running example (cf. (5) in Fig. 1).

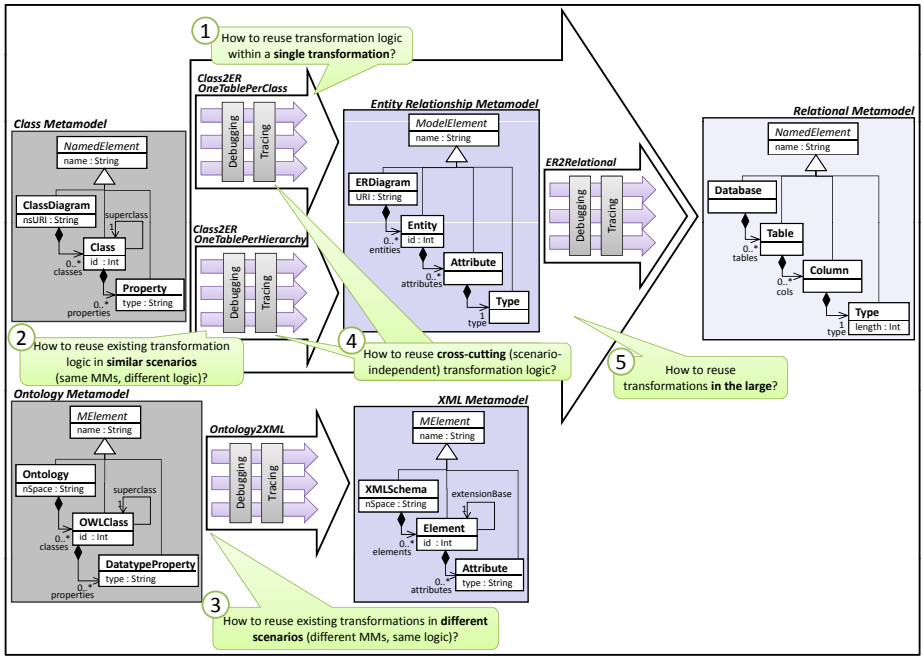

Fig. 1. Running Example - Different Scopes of Reuse

2.2 Comparison Criteria

In order to highlight commonalities as well as differences between reuse mechanisms across their scopes, a comparison framework (cf. Fig. 2) based on the common phases (i) *abstraction*, (ii) *selection*, (iii) *specialization*, and (iv) *integration* of reuse mechanisms according to [15] is proposed in the following.

Abstraction. To enable reuse, abstraction is the key of any reuse mechanism. According to [14], 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 *MMs*. Furthermore, reuse of transformation logic across platforms should be possible by generalizing from a certain *transformation language*. Abstraction by simplification allows to emphasize the information necessary for reuse, i.e., the *visible part* (e.g., interface of a function to reuse), but to hide the actual realization of the artifact, i.e., the *hidden part* (e.g., the implementation of the function) [15].

Selection. Provided that repositories of reusable artifacts exist, mechanisms are needed to efficiently find the artifacts. Such mechanisms range from *metainformation*, e.g., documentation or pre-/post-conditions, to *automatism* in the form of wizards or more advanced techniques from information retrieval [19].

Specialization. To adapt an abstracted artifact to a specific transformation, specialization is needed. Ideally, only *knowledge* of the signature of the abstracted artifact, but not of the realization is needed (i.e., *reuse in the black-box view*).

In contrast, *reuse in the white-box view* demands additional knowledge of the realization. For specialization, typically certain *mechanisms* are needed, e.g., passing of parameter values in functions or overriding/extending parts in the context of inheritance. Finally, *language-inherence* states if a transformation designer stays in the same formalism for specialization or not.

Integration. Whereas specialization solely configures an artifact, integration focuses on how reusable artifacts interact with the remaining parts of the specified transformation. Reuse mechanisms in software engineering are typically categorized into *composition* and *generation* mechanisms [3, 20]. Thereby, composition implies that integration must take place whereas 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 [17], 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.

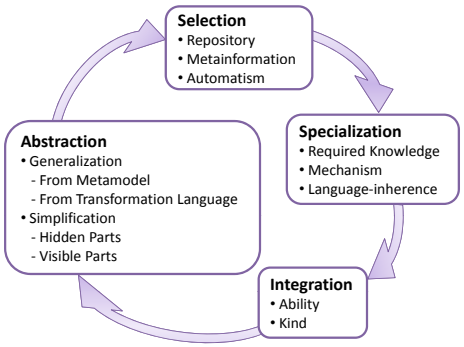

Fig. 2. Comparison Framework

3 Comparison of Reuse Mechanisms

Based on the identified scopes and the introduced comparison framework, proposed reuse mechanisms for model transformations are compared in the following (cf. Table 1 for an overview and Table 2 for details). For each reuse mechanism, representative transformation languages are listed, irrespective of their paradigm (declarative, imperative or hybrid) or scenario (e.g., inplace or model-to-model, or uni- or bidirectional transformations). To illustrate the different mechanisms, sample transformations for different facets of the running example are provided. To enhance understandability, ATL as a *single* transformation language has been used to exemplify the reuse mechanisms since it supports most of them.

3.1 Reuse of Transformation Logic within a Single Transformation

Mechanisms to avoid code duplication and thus to enhance readability and maintainability within a single transformation include *functions* and *inheritance*, since both depend on concrete MM types.

Functions. All known transformation languages provide means to extract and then reuse recurring transformation logic in functions. As can be seen in

Table 1. Categorization of Reuse Mechanisms

Scope of Reuse	Reuse Mechanisms	Supporting Representatives
Reuse of Transformation Logic within a Single Transformation	Functions Inheritance	All languages ATL, ETL, TGGs, QVT-O, Tefkat
Reuse of Transformation Logic in Similar Scenarios (same MMs, different logic)	Superimposition Transformation Product Lines	ATL, QVT, RubyTL [11], [22]
Reuse of Transformation Logic in Different Scenarios (different MMs, same logic)	Genericity DSL	SDM, VIATRA2, Tefkat, [5] External: [8], [28], Internal: [6], Epsilon, RubyTL
Reuse of Transformation Logic Independent of the Scenario	AOP HOT Reflection	Xtend All languages providing an explicit metamodel SDM, MISTRAL
Reuse of Transformation Logic in the Large	Orchestration	[12], [20], [25], ATLFlow, QVT

Fig. 3(a) which realizes the running example in a "one table per class approach", the concatenation of the `name` with `_translated` is realized by an ATL helper which is invoked several times in the transformation specification. Nevertheless, the gained *abstraction* of this reuse mechanism is low, since functions typically depend on concrete MM types, e.g., `NamedElement` in the example. Abstraction by simplification is gained since the implementation is hidden after being developed once. For *selection*, no repository exists since functions are specific to a single transformation. *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.

Inheritance. Since inheritance is employed in MMs to reuse feature definitions from previously defined classes, also inheritance between transformation rules can be applied in order to avoid code duplication. As can be seen in Fig. 3(b), rule inheritance in our running example is used in order to avoid the re-specification of the `name` assignment. Therefore, all rules inherit from the common base rule `NamedElem2ModelElem`. Nevertheless, inheritance does neither achieve *abstraction* from the actual MM types nor from the underlying transformation language, since the superrules are bound to concrete MM types. Furthermore, no abstraction by simplification takes place, since the whole implementation of the superrules is exposed to transformation designers. For *selec-*

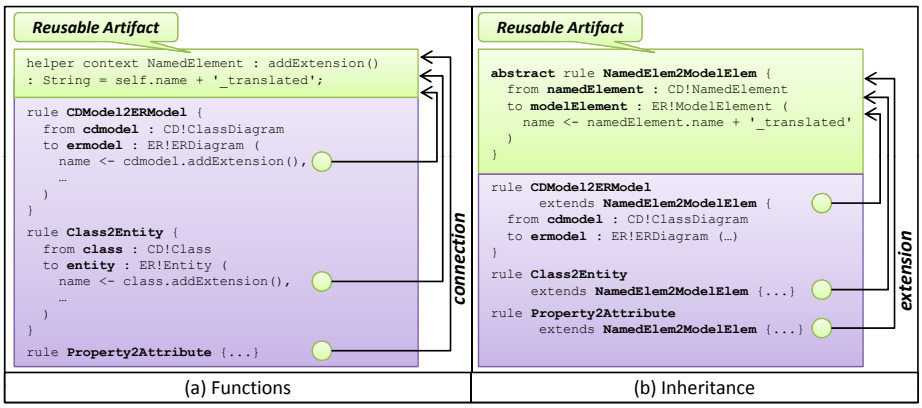

Fig. 3. Reuse Mechanisms Within a Single Transformation

tion, 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.

Synopsis. Functions as well as inheritance are both mechanisms to avoid code duplication within a single transformation. Nevertheless, they complement each other, since functions reuse arbitrary transformation logic whereas inheritance reuses assignments provided that the MM incorporates inheritance and thus allows for rule inheritance. Although inheritance is an important reuse mechanism in OOP, not all transformation languages support inheritance, or if they do, they offer different semantics as we already investigated in [31]. For example there are differences in how overridden assignments are incorporated in the overriding assignment or the way type substitutability is supported.

3.2 Reuse of Transformation Logic in Similar Scenarios

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 MMs*, but with *different transformation logic*, mechanisms are needed that allow to either alter the existing transformation, e.g., superimposition, or to configure an existing transformation such that it meets the changed requirements, e.g., transformation product lines.

Superimposition. Superimposition allows to build the union of transformation rules from different transformations. Thereby rules can be redefined, i.e., a rule is replaced by a new one if their signatures are identical, and added, whereby it is impossible to reuse the original rule. Superimposition has been proposed for ATL and QVT Relations [28] and is applicable in our running example to provide a new transformation that implements a "one table per hierarchy" approach on basis of the existing "one table per class" transformation. Thereby, the superimposed transformation redefines the rule `Class2Entity` and adds an additional helper `Closure` to calculate the transitive closure (cf. Fig. 4(a)). The so-called phasing mechanism and refinement rules in RubyTL [5] extend the idea of superimposition in the way that superimposed rules may refine the results of the original rules. Nevertheless, superimposition *abstracts* neither from the MMs (since old and redefined rules are bound to concrete MM types) nor from the transformation language. Superimposition also does not abstract by simplification, since the whole original transformation is visible to the transformation designer. Concerning *selection*, existing transformations in the "ATL Model Transformation Zoo"[3] could be reused by superimposition. 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 only. Regarding *integration*, superimposition represents again an extension-based composition mechanism.

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

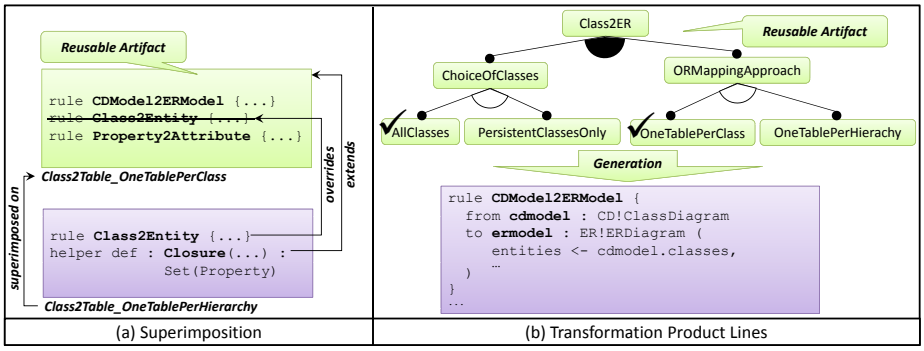

Fig. 4. Mechanisms to Reuse Transformation Logic in Similar Scenarios

Transformation Product Lines. To deal with variabilities in model transformations, approaches [12, 23] arose that allow transformation designers to explicitly specify potential variabilities in model transformations, which we call *Transformation Product Lines (TPLs)* (inspired by Software Product Lines). These approaches typically use some variability model, e.g., feature models, to guide the generation of a specific transformation. Fig. 4(b) shows a simplistic feature model for our running example, allowing to choose the classes to be translated as well as the applied OR-mapping approach. In this respect, the reusable artifact is not only the already existing transformation but 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 MM types and thus, *abstract* neither from MMs nor from the transformation language. Currently, no repository is available for *selecting* a certain TPL. *Specialization* is done by configuring the feature model, thus, no internals of the transformation are needed, being a black-box, non-language-inherent mechanism. Concerning *integration*, TPLs represent a generation-based reuse mechanism on basis of the configured feature model.

Synopsis. Superimposition as well as TPLs allow both to realize related transformation scenarios. Nevertheless, superimposition 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 develop transformations by just selecting values from the feature model. In contrast to TPLs, superimposition requires profound knowledge of the transformation language but allows flexible changes of transformations.

3.3 Reuse of Transformation Logic in Different Scenarios

Assuming that the same transformation logic should be reused in a different scenario, i.e., different source/target MMs, mechanisms are needed that allow to

decouple transformation logic from concrete MM types. In this respect, generic transformations and domain-specific languages (DSLs) have been proposed as detailed in the following.

Genericity. Genericity allows to parameterize transformation logic with types to *abstract* from concrete MMs. Thereby, approaches have been proposed for *fine-grained* genericity [18, 27], i.e., on the level of rules or functions, and *coarse-grained* genericity [6], i.e., on the level of transformations. Fig. 5 shows an example for coarse-grained genericity whereby the whole `Class2ER` transformation should be reused for an `Ontology2XML` transformation. This is possible, since the new MMs (`Ontology` and `XML`) are structurally similar to the old MMs (`Class` and `Relational`). In this case, the transformation designer only has to specify a binding model, denoting which types of the old MMs correspond to which types of the new MMs. This binding model is then used to modify the original transformation by means of a higher-order transformation (HOT) (cf. below). Thus, the transformation designer only has to care about the source/target MMs representing the visible part, whereas the implementation is hidden. Nevertheless, although the idea of generic functions and transformations is promising, no library has been established so far putting the question whether there is support for *selection* aside. Since *specialization* is done by setting type parameters in case of fine-grained genericity or specifying the binding model in case of coarse-grained genericity, it is considered as a black-box. Finally, the specialization process occurs language-inherent in case of generic functions and non-language-inherent in case of generic transformations. Whereas in case of fine-grained genericity connection is applied as a composition mechanism for *integration*, coarse-grained genericity resembles a generation-based mechanism.

Domain-specific Languages. Another way to reuse logic in different scenarios are DSLs, which provide means to simplify specification of recurring prob-

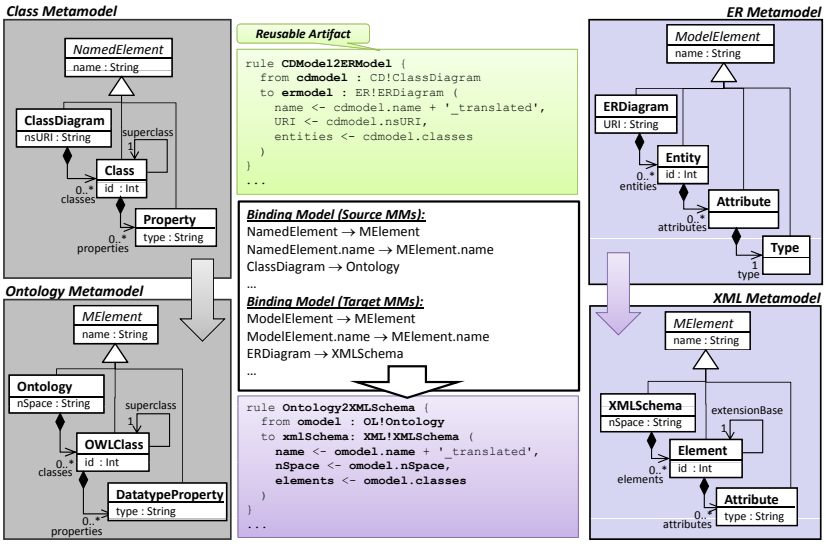

Fig. 5. Genericity On the Level of Transformations

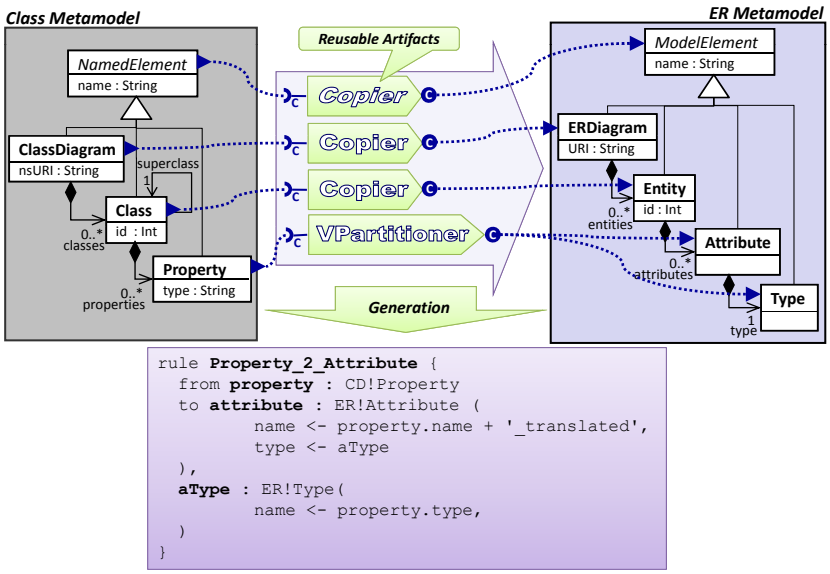

Fig. 6. External DSL Example [29]

lems in transformations. Two different kinds of DSLs can be distinguished: (i) *external* DSLs, i.e., the DSL can be used independently from the underlying transformation language, and (ii) *internal* DSLs, i.e., DSL constructs are embedded in a transformation language. External DSLs have been proposed by [9] and [29] which focus on the resolution of structural heterogeneities, e.g., the splitting of the class `Property` into the classes `Attribute` and `Type` represents a *vertical partitioning heterogeneity*. Fig. 6 shows the solution of the running example using the DSL presented in [29]. In order to execute a DSL-based specification, it has to be translated into a certain executable transformation language. Internal DSLs follow the same principles but differ in the fact that DSL constructs are tightly integrated in a certain transformation language. A representative for internal DSLs is the High Level Navigation Language (HNL) [7] which hides complex OCL navigation expressions using ATL as host language. Internal DSLs are also supported by RubyTL [8] and Epsilon[4], whereby specialized DSLs, e.g., Epsilon Flock for model migration, build upon the host language Epsilon Object Language. Although both kinds of DSLs *abstract* from concrete MMs, only external DSLs also abstract from the underlying transformation language. Concerning *simplification*, the provided DSL syntax, i.e., visible part, abstracts from the operational semantics, i.e., 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 MM types, e.g., so-called ports need to be bound

[4] `http://www.eclipse.org/gmt/epsilon`

to a certain MM element in [29] (cf. Fig. 6). Since DSL constructs are compiled to ordinary transformation code, generation based *integration* takes place.

Synopsis. Genericity as well as DSLs allow both to decouple transformation logic from concrete MM types. Genericity is a promising approach to reuse transformation logic for structurally similar MMs, either on the fine-grained level of rules or the coarse-grained level of transformations. Although especially in case of coarse-grained genericity, large parts of transformation logic are reusable, it has the drawback that it requires structural similarity resulting in a low probability for application. In contrast, DSL constructs abstract from structural similarity to a certain extent, e.g., in [29] structural flexibility is supported by providing fixed parts as well as configurable parts. Thus, although the DSL constructs do not allow to reuse whole transformations, DSLs have a higher probability for application.

3.4 Reuse of Transformation Logic Independent from the Scenario

Parts of transformation logic might be independent of any concrete scenario and might thus occur in series of transformations, e.g., cross-cutting concerns like tracing or debugging (cf. Fig. 7). To reuse such cross-cutting concerns, several mechanisms have been proposed, including higher-order transformations (HOTs), aspect-orientation (AO), and reflection.

Higher-order Transformations, Aspect Orientation and Reflection. As detailed in [25], HOTs can be applied in several ways to achieve reuse in model transformations, being (i) *transformation composition*, (ii) *transformation synthesis*, and (iii) *transformation modification*. *Transformation composition*, meaning that a HOT takes at least one transformation and potentially other configuration models as input and produces a transformation as output, can be used, e.g., to achieve genericity as described above. *Transformation synthesis*, meaning that a transformation is generated from other artifacts, is often applied in the context of DSLs to generate transformations from DSL constructs as mentioned above. Therefore, in this subsection, HOTs in the sense of *transformation modification* are covered. The HOT takes a transformation as input to, e.g., introduce cross-cutting concerns like debugging or tracing into an existing transformation. Similar goals might be achieved by AO, e.g., supported in Xtend[5] and reflection provided that the target of the reflection is the transformation itself as, e.g., in MISTRAL [16].

Considering these three mechanisms, the reusable artifact might either be the transformation, the introduced cross-cutting concerns or even both, depending on what is newly developed. All mechanisms *abstract* from concrete MM types, but none of them abstracts by simplification, since no parts are explicitly hidden. With respect to *selection*, several ATL-based HOTs are available in the Model Transformation Zoo. There are, however, no repositories for AO or reflection. *Specialization* happens typically as a black-box, provided that only transformation-independent modifications take place, e.g., for each assignment,

[5] `http://www.eclipse.org/workinggroups/oaw`

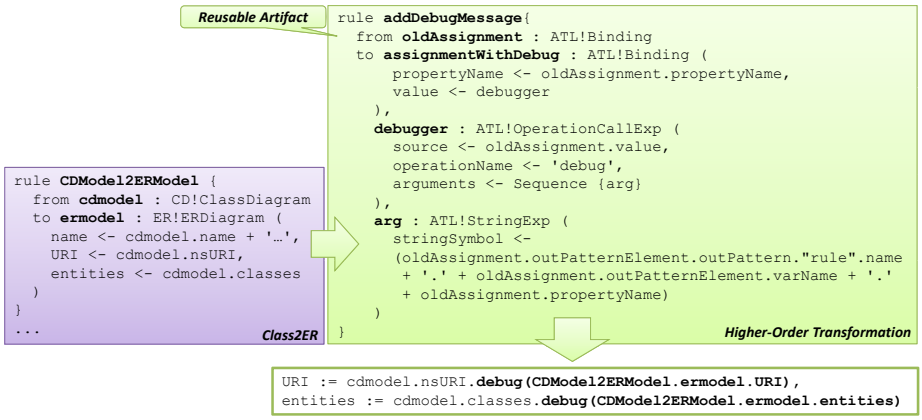

Fig. 7. Higher-order Transformations

add a debug message. The specialization mechanism is either the HOT itself, the so-called join point model in AO or the meta-rules in case of reflection [16], describing where to introduce cross-cutting code. If the to be specialized transformation and the HOT are written in the same transformation language, the HOT is considered to be language-inherent. If AO and thus the specification of the join point model is supported by the transformation language, specialization occurs language-inherent. The same is true for reflection. Concerning *integration*, all mechanisms are composition-based reuse mechanisms in terms of extension.

Synopsis. Although the three mechanisms pursue similar goals, i.e., introducing cross-cutting concerns into transformation languages without changing the underlying transformation, 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 (cf. Fig. 7). In contrast, AO allow specialization on basis of the concrete syntax and reflection on basis of the provided reflective API.

3.5 Reuse of Transformation Logic in the Large

To achieve reuse in the large, whole transformations might be reused *without adaptations*. Thus, mechanisms are needed to orchestrate model transformations, e.g., describing sequential or conditional executions of model transformations.

Orchestration. Orchestration languages have been proposed to replace low-level descriptions, e.g., in terms of Ant[6] tasks. Basically, they can be divided into approaches allowing to orchestrate model transformations written in different languages [13, 21, 26] or in a specific language only (Wires* [22], ATLFlow[7], QVT-O[8]). Fig. 8 shows a simple example in Wires* sequentially executing two ATL model transformations, first the `Class2ER` transformation and then a `ER2-Relational`) transformation. In this respect, no *abstraction* from the MMs is

[6] http://ant.apache.org/
[7] http://opensource.urszeidler.de/ATLflow/
[8] http://www.omg.org/spec/QVT/1.1/

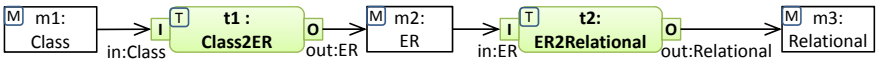

Fig. 8. Orchestration Example

achieved since the to be reused transformations still operate on concrete MMs. 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, since for orchestration only the source and target MMs of the transformations are of interest. ATL transformations might be *selected* again from the Model Transformation Zoo. Since transformations must be reused without adaptation, no *specialization* might occur. *Integration* happens by means of the orchestration language, thus it is classified as coordination.

Synopsis. Orchestration is a promising approach for reusing large portions of transformation logic. Nevertheless, the frequency of occurrence is constrained by the specificity of the reused transformations since each one is bound to concrete source and target MMs. Thus, it might be beneficial to combine orchestration with generic model transformations as proposed by [6].

Table 2. Comparison of Reuse Mechanisms

		Scope 1: Single Transformation		Scope 2: Similar Scenario		Scope 3: Different Scenario		Scope 4: Scenario-independent	Scope 5: In the Large
		Functions	**Inheritance**	**Superimposition**	**Transformation Product Lines**	**Genericity**	**DSL**	**HOT, AO, Reflection**	**Orchestration**
Reusable Artifact		*Function*	*Base Rules*	*Base Transformation*	*Feature Model, Transformation*	*Generic Rules, Transformations*	*DSL constructs, Generator*	*Base Transformation, Concern or Both*	*Transformation*
Abstraction									
Generalization	**From MM**	×	×	×	×	✓	✓	✓	×
	From TL	×	×	×	×	×	yes (external), no (internal)	×	×
Simplification	**Hidden Parts**	Implementation	none	none	Transformation	Implementation	Operational Semantics	none	implementation
	Visible Parts	Signature	all	all	Features	Signature with Type Parameters	DSL syntax	all	Signature (Source/Target MMs)
Selection									
Repository		-	-	ATL Model Transformation Zoo	×	×	DSL constructs	ATL Model Transformation Zoo (HOTs)	ATL Model Transformation Zoo
Metainfo		-	-	Documentation	-	-	Documentation, Grammar	Documentation	Documentation
Automatism		-	-	manual	-	-	semi-automatic (code completion)	manual	manual
Specialization									
Required Knowledge		black-box	white-box	black-box	black-box	black-box	black-box	black-box	-
Mechanism		parameter binding	overriding rule	redefining rules	configuration on basis of the feature model	type parameter binding	parameter binding	Transformation (HOT), Join Point Model (AO), Metarules (Reflection)	-
Language-inherent		✓	✓	✓	×	yes (fine-grained genericty), no (coarse-grained genericity)	✓	✓	-
Integration									
Ability		Composition	Composition	Composition	Generation	Composition	Generation	Composition	Composition
Kind		Connection	Extension	Extension	-	Connection	-	Extension	Coordination

4 Barriers to Model Transformation Reuse

Although numerous reuse mechanism have been proposed, barriers to model transformation reuse exists which hinder the adaptation of the mechanisms in

practice. In the following, the main barriers derived from our comparison are presented, identifying further research potentials.

Insufficient Abstraction from Metamodels. Although, some mechanisms, e.g., genericity, allow to decouple transformation logic from concrete MM types, the transformation logic is still dependent on the structure of the MMs. Thus, reuse of transformation logic between different MMs is hampered. To improve this situation, domain-specific standardized MMs would be beneficial, where standardized transformations might be defined on. Specific MMs might then extend the standardized MMs. Thus, the standardized transformations might also be reused to realize specific transformations, resembling the idea of frameworks in software engineering. This way, reuse of larger portions of transformation logic would be enabled.

Insufficient Abstraction from Platform. Except external DSLs, all reuse mechanisms target at a single transformation language, but there is little work on how to reuse transformation logic in general. A first step in this direction is presented in [30] 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 [11]. Thinking this one step further, reuse should be enabled during the whole development cycle including requirements analysis, design, implementation, and testing as also stated in [10].

Missing Repositories for Selection. As can be derived from our comparison, hardly any repository of reusable artifacts has been established so far, except the Model Transformation Zoo comprising a collection of ATL-based model transformations. This is in contrast to software engineering, where different kinds of repositories of reusable artifacts exists, ranging from fine-grained class-libraries (being delivered with any programming language) over components to coarse-grained frameworks.

Lack of Meta-information in Selection. As Table 2 reveals, there is little meta-information available for selecting a reusable artifact without having to know its internals. Therefore, it would be important to provide transformations with according meta-information, comprising source/target MMs, test models, pre- and postconditions, and documentation. Preconditions may be used, e.g., to check if input models conform to the implemented transformation logic [4]. More abstract models for model transformations, e.g., requirements, would provide an additional source of meta-information.

Challenging Specialization Mechanisms. Although most reuse mechanisms allow for specialization, they are sometimes challenging to be applied in practice. This includes especially HOTs as also stated in [24] where the user must be familiar with the abstract syntax of the transformation language. In case of inheritance, specialization has potential for improvement, since 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`). However, one important step in this direction has been the introduction of the "module" concept in transformation languages [5].

Insufficient Support for Integration in the Large. Although orchestration languages have been proposed to chain transformations to build larger ones, a main issue is the compatibility of source/target MMs between the orchestrated transformations. Thus, mechanisms are needed that ensure type compatibility in transformation chains similar to type checks in ordinary programs. This would incorporate compilation errors, if compatibility between MMs in the context of a specific transformation is violated.

5 Conclusion

In this paper, we provided an overview on proposed reuse mechanisms in model transformations. The comparison has been conducted on the basis of a framework covering the main phases in reuse, comprising abstraction, selection, specialization and integration. Although the comparison showed that a variety of mechanisms for reuse have been proposed, several barriers hindering their successful application have been identified. Furthermore, 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 make model transformation more a fact than a fiction.

References

1. A. Agrawal, A. Vizhanyo, Z. Kalmar, F. Shi, A. Narayanan, and G. Karsai. Reusable Idioms and Patterns in Graph Transformation Languages. *Electronic Notes in Theoretical Computer Science*, 127(1):181–192, 2005.
2. J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model Transformations in Practice Workshop of MoDELS'05, Montego Bay, Jamaica, 2005.
3. T. J. Biggerstaff and C. Richter. Reusability Framework, Assessment, and Directions. In *Software Reusability: Vol. 1, Concepts and Models*, pages 1–17. 1989.
4. E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL contracts for the verification of model transformations. *ECEASST*, 24, 2009.
5. J. Cuadrado and J. García Molina. Approaches for model transformation reuse: Factorization and composition. In *Proc. of ICMT'08*, pages 168–182, 2008.
6. J. Cuadrado, E. Guerra, and J. de Lara. Generic Model Transformations: Write Once, Reuse Everywhere. *Accepted for publication in Proc. of ICMT'11*, 2011.
7. J. Cuadrado, F. Jouault, J. Garca Molina, and J. Bézivin. Experiments with a High-Level Navigation Language. In *Proc. of ICMT'09*, pages 229–238. 2009.
8. J. Cuadrado and J. G. Molina. A Model-Based Approach to Families of Embedded Domain-Specific Languages. *IEEE Trans. Softw. Eng.*, 35:825–840, 2009.
9. M. Del Fabro and P. Valduriez. Towards the Efficient Development of Model Transformations using Model Weaving and Matching Transformations. *Journal on SoSyM*, 8(3):305–324, 2009.
10. E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos. TransML: A Family of Languages to model Model Transformations. In *Proc. of MODELS'10*, pages 106–120, 2010.

11. M.-E. Iacob, M. W. A. Steen, and L. Heerink. Reusable Model Transformation Patterns. In *Proc. of EDOCW'08*, pages 1–10, 2008.
12. A. Kavimandan, A. Gokhale, G. Karsai, and J. Gray. Templatized Model Transformations: Enabling Reuse in Model Transformations. Technical report, Vanderbilt University, 2009.
13. A. Kleppe. MCC: A Model Transformation Environment. In *Model Driven Architecture–Foundations and Applications*, pages 173–187. Springer, 2006.
14. J. Kramer. Is Abstraction the Key to Computing? *Commun. ACM*, 50:36–42, 2007.
15. C. W. Krueger. Software Reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.
16. I. Kurtev. Application of Reflection in a Model Transformation Language. *SoSyM*, 9(3):311–333, 2010.
17. K. Lau and T. Rana. A Taxonomy of Software Composition Mechanisms. In *Proc. of SEAA'10*, pages 102–110. IEEE, 2010.
18. E. Legros, C. Amelunxen, F. Klar, and A. Schürr. Generic and Reflective Graph Transformations for Checking and Enforcement of Modeling Guidelines. *Visual Language Computing*, 20(4):252–268, 2009.
19. A. Mili, R. Mili, and R. Mittermeir. A Survey of Software Reuse Libraries. *Annals of Software Engineering*, 5:349–414, 1998.
20. H. Mili, F. Mili, and A. Mili. Reusing software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 21(6):528 –562, 1995.
21. J. Oldevik. Transformation Composition Modelling Framework. In *Int. Conf. on Distributed Applications and Interoperable Systems*, pages 108–114, 2005.
22. J. E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo. Orchestrating ATL Model Transformations. In *Proc of MtATL'09*, pages 34–46, 2009.
23. M. Sijtema. Introducing Variability Rules in ATL for Managing Variability in MDE-based Product Lines. In *Proc of MtATL'10*, pages 39–49, 2010.
24. M. Tisi, J. Cabot, and F. Jouault. Improving Higher-Order Transformations Support in ATL. In *Proc. of ICMT'10*, pages 215–229, 2010.
25. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *Proc. of ECMDA-FA'09*, pages 18–33, 2009.
26. B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers. UniTI: A Unified Transformation Infrastructure. In *Proc. of MODELS'07*, pages 31–45, 2007.
27. D. Varró and A. Pataricza. Generic and Meta-Transformations for Model Transformation Engineering. In *Proc. of UML'04*, pages 290–304, 2004.
28. D. Wagelaar, R. Van Der Straeten, and D. Deridder. Module Superimposition: A Composition Technique for Rule-based Model Transformation Languages. *SoSyM Journal*, 9:285–309, 2010.
29. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Surviving the Heterogeneity Jungle with Composite Mapping Operators. In *Proc. of ICMT'10*, pages 260–275, 2010.
30. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Towards an Expressivity Benchmark for Mappings based on a Systematic Classification of Heterogeneities. In *Proc. of MDI'10 @ MoDELS 2010*, pages 32–41, 2010.
31. M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, D. Kolovos, M. Lauder, A. Schürr, and D. Wagelaar. A Comparison of Rule Inheritance in Model-to-Model Transformation Languages. *Accepted for publication in Proc. of ICMT'11*, 2011.