Model Transformation Reuse across Metamodels

A classification and comparison of approaches

Jean-Michel Bruel[1], Benoit Combemale[1], Esther Guerra[2], Jean-Marc Jézéquel[3], Jörg Kienzle[4], Juan de Lara[2], Gunter Mussbacher[4], Eugene Syriani[5], and Hans Vangheluwe[6,4]

[1] University of Toulouse, IRIT, France
[2] Universidad Autónoma de Madrid, Spain
[3] Univ Rennes, Inria, CNRS, IRISA, France
[4] McGill University, Canada
[5] Université de Montréal, Canada
[6] University of Antwerp, Belgium

Abstract. Model transformations (MTs) are essential elements of model-driven engineering (MDE) solutions. MDE promotes the creation of domain-specific metamodels, but without proper reuse mechanisms, MTs need to be developed from scratch for each new metamodel. In this paper, we classify reuse approaches for MTs across different metamodels and compare a sample of specific approaches – model types, concepts, a-posteriori typing, multilevel modeling, and design patterns for MTs – with the help of a feature model developed for this purpose, as well as a common example. We discuss strengths and weaknesses of each approach, provide a reading grid used to compare their features, and identify gaps in current reuse approaches.

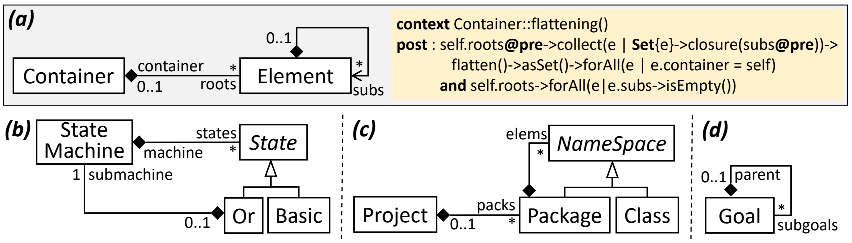
Keywords: Model Transformation, Reuse, Classification, Feature Model, Model Types, Concepts, A-posteriori Typing, Multilevel Modeling, Transformation Design Patterns

1 Introduction

As model-driven engineering (MDE) is used for engineering evermore numerous and complex systems, model transformations (MTs) are becoming more and more complex pieces of software. Like for any other piece of software [1], *reuse mechanisms* for MTs have been proposed to limit reimplementing a transformation from scratch every time a new but related need arises. In this paper, we focus on the reuse of MTs that were developed for a particular metamodel, but are then applied to models typed by other metamodels, i.e., reuse *across* metamodels.

Many use cases of MT reuse have been identified in the literature [2], providing useful classifications. Since the use cases of MT reuse imply very different trade-offs among non-functional properties such as type-safety, performance, expressiveness and user-friendliness, no single MT reuse approach fits them all.

In this paper, we propose a classification of MT reuse approaches that work across metamodels, and compare a sample of specific approaches—namely model types [3,4],



```
context Container::flattening()
post : self.roots@pre->collect(e | Set{e}->closure(subs@pre))->
          flatten()->asSet()->forAll(e | e.container = self)
        and self.roots->forAll(e|e.subs->isEmpty())
```

Fig. 1: (a) Reusable model transformation scheme. (b, c, d) Metamodels for which the model transformation wants to be reused.

concepts [5,6], a-posteriori typing [7], multilevel modeling [8], and design patterns for MTs [9]—with the help of a feature model developed for this purpose, and a common example. We discuss strengths and weaknesses of each proposal, provide a reading grid to compare their features, and identify gaps in current reuse approaches.

The paper is organized as follows. Section 2 motivates the need for reuse mechanisms across metamodels and presents a running example. Section 3 defines classification criteria using a feature model. Section 4 compares five existing approaches based on the classification and the running example, and Section 5 discusses trade-offs. Section 6 overviews related classification attempts and reuse techniques, and Section 7 concludes by identifying challenges for the MT community.

2 Motivation

MDE supports the creation of metamodels to describe models using the most appropriate primitives and level of abstraction. However, this entails the creation of all kinds of services for each metamodel, including MTs. Without proper reuse mechanisms, MTs need to be created from scratch even if there are MTs with the same goal but defined over similar yet different metamodels.

As a concrete example, consider a MT that implements the common *flattening* operation. This MT traverses a given hierarchy and extracts its elements into a flat collection. Fig. 1(a) illustrates a specification for such a MT, defined over a minimal metamodel that contains just the elements the MT needs (Container and Element). In practice, the MT would be implemented using languages like ATL [10], ETL [11], or Kermeta [12], but to stay language-agnostic, we only show a post-condition that identifies its effect. The first two lines of the postcondition state that, for a given hierarchy, all (sub-)elements should become contained in the same root container; the last line ensures the hierarchy is removed.

Flattening is recurrent in many contexts, like in structural modeling (class/package hierarchies, goal hierarchies) and behavioral languages (state machines, activity diagrams). Figs. 1 (b), (c), (d) show three typical metamodels of these kinds of languages.

Without proper reuse mechanisms, a flattening MT needs to be implemented from scratch for each metamodel. Some ad-hoc reuse approaches are applied in practice, like *clone-and-own* (copy-paste and manual adaptation) or translating the models of interest to the metamodel accepted by the reused MT. Neither approaches are optimal.

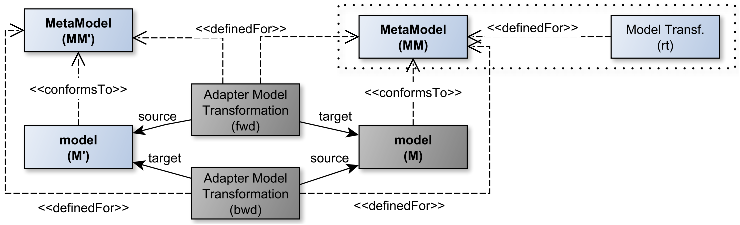


Fig. 2: Explicit model adaptation approach to MT reuse

In the first case, manual adaptation is time-consuming, error-prone, hardly scalable, and leads to well-known maintenance problems with code clones [13]. In the second case, illustrated by Fig. 2, an existing MT (*rt* on the right) defined for a metamodel MM, wants to be reused on a model (M' on the left) conformant to a different metamodel MM'. In this figure (and following ones), light boxes represent existing artifacts, and dark ones represent new artifacts to be built. An *adapter* transformation is required to translate the model into one that conforms to the metamodel the reused transformation conforms to, so that the MT can be applied to this new model M. This is not efficient since it requires executing an additional transformation in addition to the reused one. Moreover, a reverse MT may be needed to transform the result back to the original model's metamodel.

The MT community has proposed several approaches to facilitate reuse across metamodels, like model typing, a-posteriori typing, concepts, multilevel modeling and transformation patterns, among others [14,15,16,17]. These approaches have different tradeoffs and are applicable in different scenarios and contexts. Hence, there is an urging need to classify and compare them to know which approach to use in a given situation.

3 Classification

We introduce a feature model to classify the different alternatives for MT reuse across metamodels. The model, shown in Figs. 3 and 4, presents the features of the reuse mechanism as well as properties of the reused transformation. In the following, we write rt to denote the MT to be reused.

Strategy. In a *systematic* reuse strategy, a MT is developed by reusing specific units that were made available a priori. This is analogous to software built following a component-based design. In this case, rt was developed with the intention of being reused. Hence, depending on the reuse approach, the MT needs to be packaged as a component [6], as a pattern [9], or the metamodel the MT is defined on needs to be sliced [18]. All other kinds of reuse are considered *opportunistic*.

Mappings. A reusable transformation rt, defined over a metamodel MM, is applicable to a number of different metamodels MM'. The way to specify the correspondences or mappings between MM and MM' depends on the reuse approach, and determines the set of metamodels where rt can be reused. Fig. 4 shows the alternative features for mapping specifications.

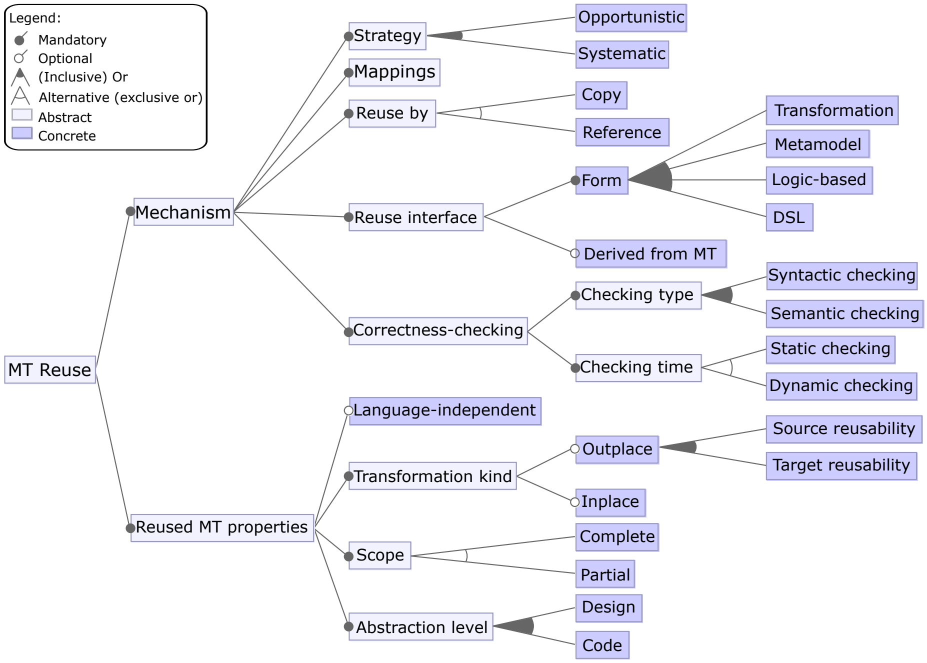


Fig. 3: Feature model: mechanisms for reuse and scenarios of reuse (the *Mapping* feature is expanded in Fig. 4

- **Arity**: The relation between MM and the new reuse context MM' can be *one-to-one*: injective where each element in MM needs to be mapped to exactly one element in MM'. The mapping can be *one-to-many*: each MM element is mapped to any number of MM' elements, including none. It can also be *many-to-one*: an MM element can be mapped several times. Finally, the most general kind of mapping is *many-to-many*: elements in both MM and MM' can be mapped several times.
- **Style**: The objects over which rt are reused can be specified either by *extension* (i.e., enumerating them) or by *intension* (i.e., providing necessary and sufficient conditions that characterize the objects). Moreover, intensional specifications can be evaluated statically at *compile-time*, *dynamically* at run-time, or at the convenience of the user (*user-defined*).
- **Level**: *Intra-level* mappings relate elements at the same metalevel: either two *meta*-models, which is the most common case, or two *models*. In contrast, mappings *across* levels relate elements at different metalevels by means of *instantiation* (e.g., in multilevel modeling) or *typing* relationships (e.g., in transformation patterns, where rule elements are typed w.r.t. a metamodel).
- **Definition**: The mapping between MM and MM' can be *explicit*, i.e., defined by the user (using either an extensional or intensional approach), or be *inferred* automatically, e.g., using name matching [3] or structural similarity criteria [14].

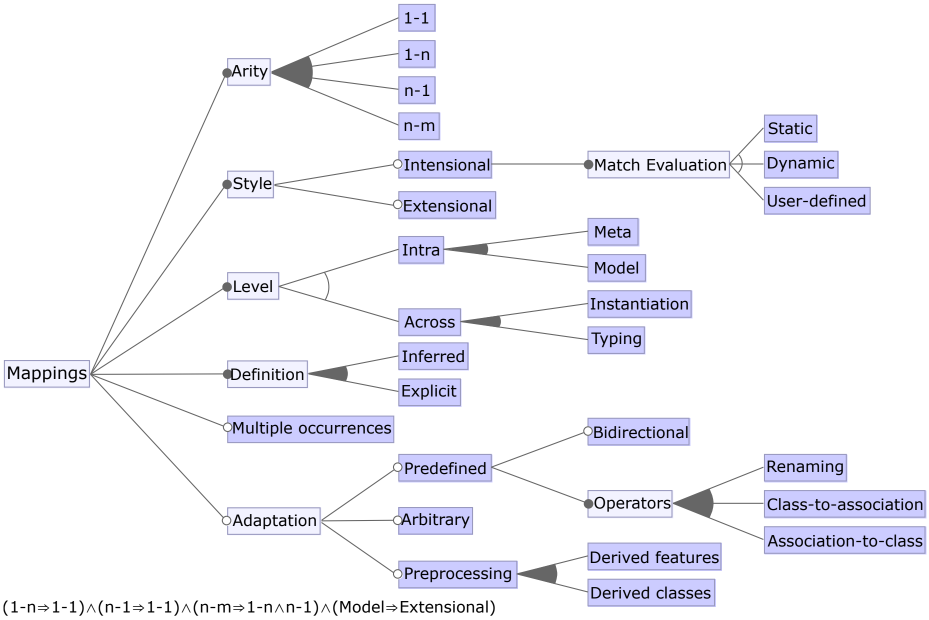


Fig. 4: Feature model: specification of mappings

- **Multiple occurrences**: This refers to the possibility to define multiple application contexts for rt within a metamodel MM', all of which are handled simultaneously by rt, perhaps using a composition mechanism for coordination. Most existing approaches only support one application context at a time.
- **Adaptation**: To widen the number of metamodels where a transformation can be reused, several mechanisms bridge heterogeneities between MM and MM'. Some approaches provide a set of *predefined* operators for specific kinds of adaptations, such as *renaming* a class, mapping a *class to* an *association*, or mapping an *association to* a *class* [6,14] (please note that our feature model does not list all possible predefined adaptation operators). Such operators may be *bidirectional* or not. Other approaches allow *arbitrary* adaptations between MM and MM', usually defined by means of OCL expressions. It is also possible to rely on a *preprocessing* step that adds the necessary *derived classes* or *derived features* to MM', making it structurally similar to MM and allowing a direct mapping between them, before applying rt [19,6].

Reuse by. This feature refers to whether the original transformation is *copied* or *referenced*. In the *clone-and-own* approach (cf. Section 2), the developer reuses a copy of rt in the transformation. Therefore, any updates to rt will not be propagated to the new transformation. Instead, the *adapter* approach of Fig. 2 reuses rt by reference, and hence any further update to the transformation affects all places where it was reused.

Reuse interface. Reusable transformations expose an interface for reuse that can take different *forms* depending on the approach. It can be a *metamodel* declaring the nec-

essary classes and features in the context of reuse [3,4,9,6], a *logic-based* specification stating the constraints that a metamodel should fulfill to ensure a correct MT reuse [17], or a model describing metamodel requirements using a domain-specific language (*DSL*) [15]. Sometimes, this reuse interface can be (semi-)automatically *derived from the MT* [17,15,18]. While the above-mentioned interface kinds yield a black-box approach to reuse, the interface for reuse in white-box approaches is the reusable MT or an abstraction of it [9,14]. This is appropriate when a larger MT is to be composed out of smaller fragments. Both interface kinds can be combined.
Correctness checking. Different approaches make different choices on how and when the correctness of rt with respect to m' and MM' should be checked.

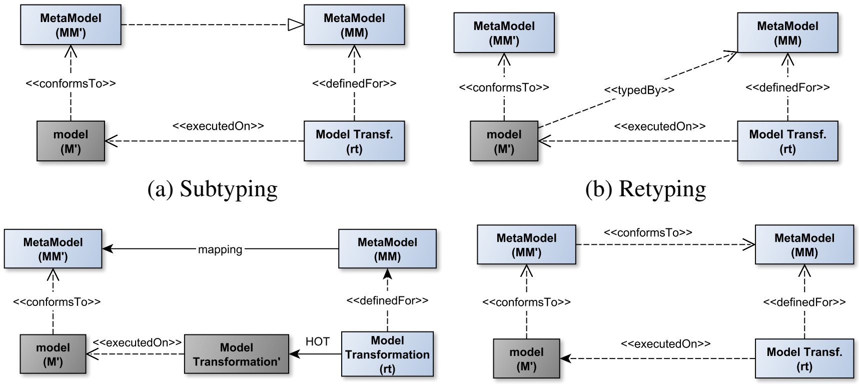
- **Checking-Type**: Checking can be either *syntactic*, e.g., simple type checking, or *semantic*, typically also verifying the satisfaction of well-formedness rules expressed in OCL, or additional semantic conditions capturing the transformation *intent* (e.g., like bisimulation relations) [20].
- **Checking-Time**: When the correctness of rt is checked *statically*, it is ensured that it will be syntactically correct for all models conforming to the new context of reuse MM'. Instead, a *dynamic* check needs to inspect at run-time that every (read/write) access to the model by rt is correct. Static checking of semantic properties requires some form of theorem proving or model checking, while dynamic checking only requires a run-time evaluation of OCL constraints.

Properties of reused transformation. Transformation reuse approaches can be *language-independent* (i.e., the reusable transformation can be written in any transformation language) or be specific for a transformation language (e.g., ATL or graph transformation). Moreover, some approaches may be limited to a particular kind of transformation, application scope or abstraction level.

- **Transformation kind**: The reused transformation can be either *inplace* or *outplace* (i.e., model-to-model). In the former case, the mechanism needs to ensure that write accesses to the model are correct. In the latter case, the new context of reuse can be for the source metamodel, which is typically read-only (*source reusability*), for the target metamodel, which is typically write-only (*target reusability*), or for both.
- **Scope**: The reused unit can be a *complete* model transformation or a part of it, e.g., a rule (*partial*).
- **Abstraction level**: Reuse can be at the *design* level, e.g., in the form of design patterns [9], or directly at the implementation level to reuse transformation *code*.

4 Comparison of Some Existing Approaches

In this section, we analyze five prominent reuse approaches, classifying them by the introduced feature model. Each approach is based on a different technique, summarized in Fig. 5. Model types (Fig. 5a) is based on establishing a subtyping relation between metamodels. A-posteriori typing (Fig. 5b) works by retyping the model so that the reused MT can be applied to it. Concepts rely on genericity to rewrite the MT using a high-order transformation (Fig. 5c) to make it applicable to a particular metamodel.



(a) Subtyping

(b) Retyping

(c) High-Order Transformation/Generative

(d) Multilevel

Fig. 5: Different techniques enabling MT reuse across metamodels

Similarly, MT patterns use a generative approach to synthesize specific MT code from a design pattern. Finally, multilevel modeling exploits the typing relation to apply the MT two (or more) metalevels below (Fig. 5d).

Table 1 summarizes how each approach instantiates the feature model. We provide more details on their working scheme using the running example in what follows.

4.1 Model Typing

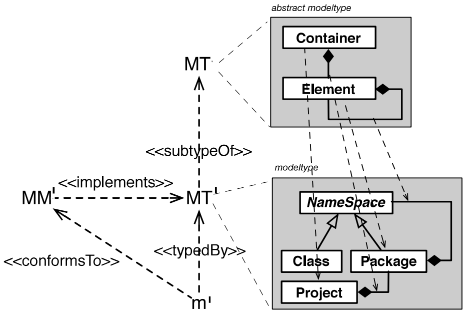


Fig. 6: Reuse with model typing

Model Types were introduced by Steel et al. [3], as an extension of object typing to provide abstraction from object types and enable model manipulation reuse. The type of a model is a set of types of objects that may belong to the model, and their relationships. While a model conforms to one and only one metamodel (the one containing all the types needed to instantiate objects of the model), it can have several model types which are subsets of its metamodel. Substitutability is the ability to safely use a model of type A where a model of type B is expected. Substitutability is supported in the model type theory by defining a subtyping relationship among model types [4,21,22].

Fig. 6 illustrates model typing, showing how to reuse the flattening MT defined on MT for the object-oriented metamodel MM'. Based on derived attributes defined within the object-oriented metamodel, if an isomorphism is statically (or possibly) found, the flattening MT can be safely applied on the instances of the object-oriented metamodel (m'). Melange employs adapter generators at compile time to ensure the

Table 1: Classification of MT reuse approaches

Feature	**Model-Typing**	**Concepts**	**A-posteriori**	**Multilevel**	**MT Patterns**
Mechanism					
Strategy	Systematic Opportunistic	Systematic Opportunistic	Systematic Opportunistic	Systematic Opportunistic	Systematic
Reuse by	Reference	Copy	Reference	Reference	Copy
Reuse interface	Metamodel Can be derived	Metamodel Can be derived	Metamodel	Metamodel	Transformation
Checking type	Syntactic Semantic (pre. and post.)	Syntactic	Syntactic	Syntactic	Syntactic
Checking time	Static (type-level) Dynamic (inst-level)	Static	Static (type-level) Dynamic (inst-level)	Static	Static
Mechanism.Mappings					
Arity	1-1, 1-n, n-1, n-m[a]	1-1, 1-n, n-1, n-m	1-1, 1-n, n-1, n-m	1-1, 1-n	1-1
Style	Extensional	Extensional	Extens. (type-level) Intens. (inst-level) Dynamic match	Extensional	Extensional
Level	Intra/meta	Intra/meta	Intra/meta	Across/ instantiation	Across/ typing
Definition	Inferred	Explicit	Explicit	Explicit	Explicit
Multiple occur.	no	no	no	no	no
Adaptation	Renaming, derived feats	Renaming, c-to-a a-to-c, arbitrary, derived feats, derived classes	Renaming, arbitrary, bidirectional, derived feats	Renaming, derived feats	Renaming
Reused MT properties					
Lang. indep.	yes	no	yes	yes	yes[b]
Transf. kind	Inplace Outplace (M2M & M2T)	Inplace [5] Outplace [6] src/tar reusability	Inplace Outplace src/tar reusability	Inplace Outplace src/tar reusability	Inplace Outplace src/tar reusability
Scope	Complete	Complete	Complete	Partial[c]	Partial
Abstrac. level	Code	Code	Code	Code	Design

[a] Preprocessing of derived features for alignment

[b] By additional code generators

[c] Through refining transformations [5]

adaptation at runtime of the actual application of the MT on the instances of the targeted metamodel [22].

4.2 Concepts

Inspired by generic programming, concepts were proposed in [5] as a mechanism to express requirements for generic model management operations and transformations. A concept is similar to a metamodel, but its elements are parametric types that need to be bound to elements in a metamodel. Generic transformations are defined over concepts. When a concept is bound to a metamodel, the associated transformation gets rewritten in terms of the metamodel and can be applied to its instances. In this approach, adapters [6] enable more flexible bindings by the use of OCL expressions in mappings, which get injected in the rewritten MT code.

Fig. 7 shows how to reuse the flattening MT for the object-oriented metamodel using concepts. The flattening metamodel is considered the concept, whose elements need to be bound to elements in the concrete metamodel. In this case, an adapter is needed to

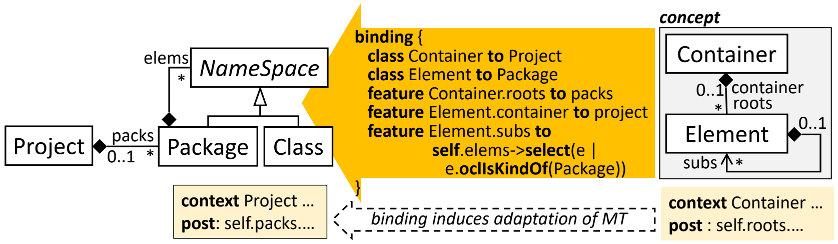


Fig. 7: Binding of concept to metamodel, and MT adaptation (sketch)

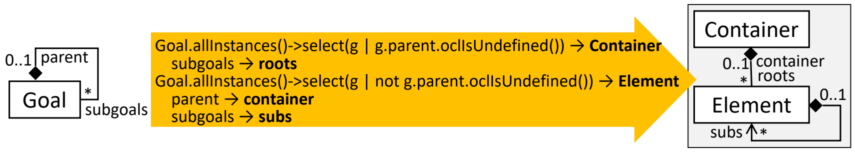

0..1
parent
Goal
*
subgoals
Goal.allInstances()->select(g | g.parent.oclIsUndefined()) → Container
subgoals → roots
Goal.allInstances()->select(g | not g.parent.oclIsUndefined()) → Element
parent → container
subgoals → subs
Container
0..1
container
*
roots
Element
0..1
subs
*

Fig. 8: A-posteriori instance-level specification for the flattening of goal models

filter Class objects out of the elems relation (see last line of binding). As a last step, the generic transformation is rewritten using the bindings and the adapters.

4.3 A-posteriori Typing

A-posteriori typing [7] permits classifying objects by classes different from the ones used to initially create the objects, and hence enables multiple, partial, dynamic typings. This approach allows opportunistic reuse as MTs defined for a metamodel can be reused with other models after being reclassified. In this way, MTs become highly reusable as, similar to Java interfaces, one can design metamodels whose goal is not object creation, but to serve as a type for MTs. Fig. 5b shows the working scheme of this approach: a model typed by an arbitrary metamodel is assigned new types from the metamodel a MT was defined on, and as a result, the MT can be executed as-is on the model.

A-posteriori typing specifications can be type-level or instance-level. The former induces a static relation between two metamodels, so that instances of one can be seen as instances of the other. This mapping style is similar to those in model typing. Instance-level specifications are more expressive than type-level ones, as they permit classifying objects by queries that assign a given type to the result of the query. This typing is dynamic because classification may depend on the run-time value of the object properties, which may evolve. Moreover, it allows an object to have multiple a-posteriori types.

Fig. 8 shows an instance-level a-posteriori specification to reuse the flattening transformation with goal models. In particular, all Goal objects with no parent are retyped as Containers, all Goal objects with a parent goal are retyped as Elements, and references are also retyped properly. When a goal model gets retyped by this specification, the MT can be applied as-is on the model. This instance-level example that partitions Goal objects into two sets at run-time illustrates the power of dynamic match evaluation, which among the surveyed approaches is only supported by a-posteriori typing.

4.4 Multilevel Modeling

Multilevel modeling was proposed in [23] as a way to enhance flexibility in modeling by enabling an arbitrary number of metalevels and a dual type/instance facet for model elements, so that they are instances with respect to the metalevel above, and types with respect to the metalevel below. This approach facilitates the definition of domain-specific metamodeling languages and families of languages [8], which can be iteratively refined in successive metalevels to account for domain-specific aspects. Model management operations defined in upper metalevels become generic and applicable to the instances in direct and indirect lower metalevels.

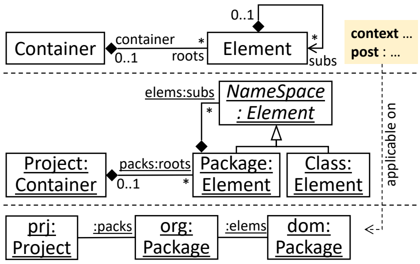
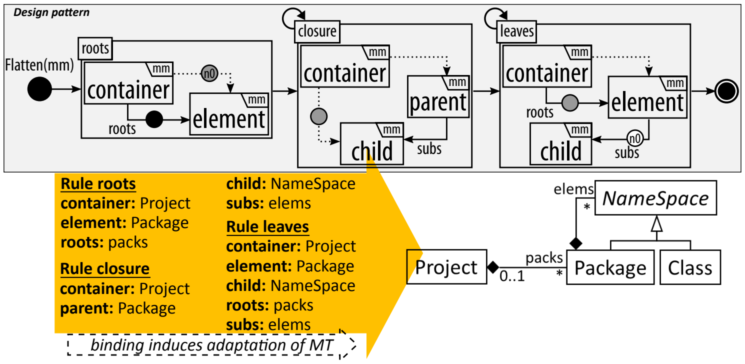


Fig. 9: Reuse by multilevel modeling

Fig. 9 uses multilevel modeling to reuse the flattening transformation with a metamodel for object-oriented design. The metamodel of the flattening transformation needs to be promoted to a higher metalevel, and the object-oriented design metamodel needs to be created as an instance of it. In this way, the transformation can be applied on the object-oriented models created in the lower metalevel.

4.5 Design Patterns for Model Transformations

Design patterns are artifacts reputed for reuse in software engineering. Unlike the previous approaches, reuse must be planned for at design-time. The approach in [9] introduces a DSL, called DelTa, to define design patterns for MTs. Given a pattern in DelTa, a higher-order transformation (HOT) synthesizes a partial MT that implements the pattern in a dedicated MT language by means of code generation. A DelTa model describes an ordered set of rules containing abstract entities and relations that can be matched (positively or negatively), created, or deleted.



Rule roots
container: Project
element: Package
roots: packs

Rule closure
container: Project
parent: Package
child: NameSpace
subs: elems

Rule leaves
container: Project
element: Package
child: NameSpace
roots: packs
subs: elems

Fig. 10: Binding of flattening design pattern to metamodel

Table 2: Comparison of model transformation reuse approaches

	Model-Typing	**Concepts**	**A-posteriori**	**Multilevel**	**MT Patterns**
Reusing existing MT (opportunistic)	slicing	slicing	free	promotion	N.A.
Making a MT reusable (systematic)	abstract MM	generic MM (concept)	role MM	deep MM	design pattern
Customization technique	adapter + mappings	adapter + mappings	adapter + mappings	instantiation	mappings
Customization ease	low to high	low to high	low to high	medium to high	high
Customization expressiveness	high (polymorphic reuse)	medium-high (parametric reuse)	very high (multi-matching, dynamic)	medium (instantiation)	low (limited matching)
Execution cost overhead	evaluation of adapter at run-time	none (adapter injected in MT at compile time)	evaluation of adapter at run-time	traverse typing relations at run-time	none (MT excerpt generated from pattern)
Property preservation guarantees	static typing, polymorphism	static typing, generics	dynamic typing, constraint solving	static typing, multilevel	static typing, generative

The top of Fig. 10 shows a design pattern in DelTa representing the flattening operation that satisfies the specification in Fig. 1. It consists of three rules that must be applied in this order on a given metamodel mm. It is thus an inplace transformation. The roots rule creates a trace link (dotted arrow) from the container to the root elements and removes the roots relation. In DelTa notation, elements in gray shall be created, those in black shall be removed, and all others are part of the constraint that shall be matched. Elements labeled with n0 are part of the negative constraint that shall not be matched. The closure rule creates a trace link from the container to all sub-elements recursively (i.e., the transitive closure). The leaves rule creates a roots relation from the container to all elements with no sub-element. The Flatten design pattern and the mapping are specified independently from the MT language. However, the HOT generates its implementation in a specific MT language for a specific metamodel.

Using the notation in Fig. 5c for the MT patterns approach, MM corresponds to the metamodel of DelTa (see [9]), rt is the Flatten design pattern, and MM' is the object-oriented design metamodel in this example. Then, similar to the concepts approach, rt is reused by generating a MT tailored to MM'.

5 Discussion

From the configurations shown in Table 1 for several MT reuse approaches, next, we discuss their differences with regards to a number of properties: if reuse is opportunistic or systematic, the customization techniques used to adapt a MT to a particular context, the customization ease and expressiveness, the overhead at execution time, and the properties guaranteed by the approaches. Table 2 synthesizes the results.

To reuse a MT, it is first necessary to make it reusable. This can be done a priori when the MT is defined (i.e., systematic reuse) or a posteriori when the MT is reused (i.e., opportunistic reuse). Model typing, concepts, a-posteriori typing and multilevel modeling support both kinds of reuse. For opportunistic reuse, the former two provide slicing mechanisms to extract the relevant part of the metamodel used by the MT [18], and for planned reuse, they support the definition of the MT on a generic metamodel (called *abstract* in model typing and *concept* in the concepts approach) which is the minimal metamodel the MT requires. Multilevel modeling uses promotion (i.e., pulls a metamodel one metalevel up) to handle opportunistic reuse, and it creates deep metamodels (i.e., which can be instantiated in successive metalevels) for systematic reuse. In a-posteriori typing, there is no specific technique to simplify opportunistic reuse, while for systematic reuse one can create a role metamodel [7] (i.e., its primary goal is not instantiation but retyping). Patterns are only relevant for systematic reuse, where abstract patterns are made available to be applied on a specific metamodel.

Once the MT rt is available for reuse, it is necessary to align the initial metamodel MM over which it is defined, to the actual metamodel MM' on which it is to be reused. Model typing, concepts, a-posteriori typing, and patterns rely on syntactic mappings. When further customizations are required to apply rt in a particular context, model typing, concepts, and a-posteriori typing also support the definition of explicit adapters. Multilevel modeling relies on instantiation to map the initial metamodel MM to the actual metamodel MM' one metalevel below. In the case of patterns, the developer must typically refine the MT by hand if the mapping is complex.

The complexity of the adapters depends on the syntactic distance between the initial and actual metamodels. The cost to specify them can range from low to high accordingly. Multilevel modeling requires a special metamodeling architecture, and patterns require an explicit definition of the mapping even in case of an isomorphic alignment, while other approaches may infer it automatically.

Regarding the expressiveness of the mapping customization, model-typing relies on polymorphic reuse and concepts on parametric reuse. A-posteriori typing supports in addition multi-matching (i.e., a model element can get several a-posteriori types) and dynamic typing. Multilevel modeling uses instantiation for customization, and patterns are limited to isomorphic matching.

The expressiveness for defining the customization comes with the cost of its evaluation when the MT is reused. Model typing and a-posteriori typing evaluate the adapters when the MT is called, and multilevel modeling follows a similar approach by traversing the typing relationships at run-time. However, the added flexibility of a-posteriori typing for instance-level specifications may incur run-time penalties, as object types are dynamically calculated by queries. The concepts approach evaluates the adapters at compile-time to generate a new MT fitting the new metamodel MM'. The execution cost is not applicable for patterns since they are reused at design-time [9], and then compiled into a specific MT language.

Finally, the property preservation guarantee relies on the underlying theory used by each approach. At design-time, model typing relies on polymorphic reuse, concepts rely on parametric reuse, multilevel modeling relies on deep instantiation, and patterns use a generative approach. A-posteriori typing uses constraint solving at design-time to discard potentially unsafe matchings, but the correctness guarantees are limited by the bounded search of the constraint solver [7].

Altogether, the discussed approaches cover most features in the feature model, but a few remain uncovered. Two specification styles are not favored by any approach. First, with respect to intensional specification of mappings, they are either evaluated statically (in model types) or dynamically (in a-posteriori typing); however, having user-defined evaluation points in the transformation execution is unexplored. As for the level of mappings, they are either across levels (instantiation for multilevel modeling, and typing for patterns) or intra-level between metamodels (the rest); however, no approach supports intra-level mappings between models. This latter specification style could be realized by mapping the model elements to be transformed to the elements in reused rules, which would lead to highly customized but very costly reuse specifications.

Other uncovered options relate to the functionality offered by the reuse mechanism. First, supporting semantic checkings (i.e., in line with the so-called transformation "intents" [17,24]) would be a way to further characterize correct reuse contexts by expressing requirements on the expected (possibly dynamic) semantics of the reuse context. To our knowledge, there is no approach enabling the definition or checking of MT intents. Another uncovered feature is supporting multiple occurrences (i.e., reusing several instances of a MT). This would need mechanisms for composing and synchronizing the multiple MT occurrences, in line with "localized transformations" [25] or "flexible instantiation policies" [26]. More generally, automated mechanisms for composing a MT out of reused partial MTs are not exploited by the analyzed approaches. This is so as all approaches – except patterns – see the reused MT as a black box. In patterns, one can manually compose reused MTs, but none of the approaches have facilities to automate the composition process at the code level. That would require a combination with internal composition techniques like [27,28].

6 Related Work

Reuse of MDE-related artefacts, like metamodels [5] and DSLs [8,22,29], is being actively investigated. In this paper, we have focused on reuse of transformations across metamodels, so-called inter-transformations in [2]. Other kinds of MT reuse include intra-transformation reuse (i.e., reuse within a MT for the same metamodel) and transformation composition. We refer to [2] for further details on these kinds of reuse.

Intra-transformation reuse is typically specific for a transformation language. Some of the proposed techniques include rules with variability [30], ATL module superimposition [31], and rule inheritance [32]. Other internal composition mechanisms are phases, hooks [27] and unit combinators [28]. As mentioned in Section 5, an interesting line of work is the combination of inter- and intra-transformation reuse.

Several classifications of MT approaches [33] and tools [34] exist. The features of some MT approaches, like *parameterization* or support for high-order transformations, facilitate reuse. Most reuse approaches are independent of the MT language. However, those that are dependent (like concepts [6]) benefit from the declarative style of the MT language, as it simplifies the rewriting of the MT specification.

For space constraints, we left out a detailed comparison with other reuse approaches across metamodels, like [14,15,17,16]. Anyhow, these approaches were taken into account when developing the proposed feature model. Mapping operators [14] are predefined adapters between metamodels, which by themselves define a MT. In [15], a transformation requirement model is extracted from an existing MT to describe the metamodels over which the MT can be reused. This is similar to constraint-based model types [17], but while requirement models use a DSL to express typing requirements, constraint-based model types use logic. Finally, generic MTs [16] are similar to concepts, but specifying relations between the type parameters is not possible, and there is limited support for adaptation [16]. For comparison, we provide the feature model configuration of those approaches at `http://bit.ly/bellairs18`.

7 Conclusion and Perspectives

To achieve true engineering of MDE solutions, mechanisms to scale up MT to industrial practice – like reuse – are required. In this paper, we have analyzed and classified approaches to *MT reuse across metamodels* in order to clarify the existing reuse options. We have provided a feature model mapping the current option space, and identified gaps that signal opportunities for further research and challenges for the MT community. These include the specification and checking of advanced semantic properties indicating a correct reuse [17], and the combination of intra- and inter-transformation reuse approaches.

In the future, we would like to outline guidelines for selecting the appropriate reuse technique depending on the scenario. We also plan to expand our classification with a goal model to facilitate the decision on the reuse choice, and to open the spectrum to other reuse scenarios. Analyzing how often are MTs reused in practice and detecting reuse opportunities, e.g., using tools like [35], remain as future work.

Acknowledgements. Work partially supported by the Spanish MINECO (TIN2014-52129-R), the R&D programme of the Madrid Region (S2013/ICE-3006), the Safran/Inria/CNRS collaboration GLOSE, and the Inria/CWI Associated Team ALE.

References

1. Krueger, C.W.: Software reuse. ACM Comput. Surv. **24**(2) (1992) 131–183
2. Kusel, A., et al.: Reuse in model-to-model transformation languages: are we there yet? SoSyM **14**(2) (2015) 537–572
3. Steel, J., Jézéquel, J.M.: On model typing. SoSyM **6**(4) (2007) 401–414
4. Guy, C., Combemale, B., Derrien, S., Steel, J., Jézéquel, J.M.: On model subtyping. In: ECMFA. Volume 7349 of LNCS., Springer (2012) 400–415
5. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. SoSyM **12**(3) (2013) 453–474
6. Cuadrado, J.S., Guerra, E., de Lara, J.: A component model for model transformations. IEEE Trans. Software Eng. **40**(11) (2014) 1042–1060
7. de Lara, J., Guerra, E.: *A Posteriori* typing for model-driven engineering: Concepts, analysis, and applications. ACM TOSEM **25**(4) (2017) 31:1–31:60
8. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific metamodelling languages. SoSyM **14**(1) (2015) 429–459
9. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Computer Languages, Systems & Structures **46** (2016) 106–139
10. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1-2) (2008) 31–39
11. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: ICMT. Volume 5063 of LNCS. (2008) 46–60
12. Jézéquel, J., Barais, O., Fleurey, F.: Model driven language engineering with Kermeta. In: GTTSE. Volume 6491 of LNCS., Springer (2011) 201–221

13. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: ICSE, IEEE Computer Society (2009) 485–495
14. Wimmer, M., et al.: Surviving the heterogeneity jungle with composite mapping operators. In: ICMT. Volume 6142 of LNCS., Springer (2010) 260–275
15. de Lara, J., di Rocco, J., di Ruscio, D., Guerra, E., Iovino, L., Pierantonio, A., Cuadrado, J.S.: Reusing model transformations through typing requirements models. In: FASE. Volume 10202 of LNCS., Springer (2017) 264–282
16. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: UML. Volume 3273 of LNCS., Springer (2004) 290–304
17. Zschaler, S.: Towards constraint-based model types: A generalised formal foundation for model genericity. In: VAO@STAF, ACM (2014) 11–18
18. Cuadrado, J.S., Guerra, E., de Lara, J.: Reverse engineering of model transformations for reusability. In: ICMT. Volume 8568 of LNCS., Springer (2014) 186–201
19. Diskin, Z., Maibaum, T.S.E., Czarnecki, K.: Intermodeling, queries, and kleisli categories. In: FASE. Volume 7212 of LNCS., Springer (2012) 163–177
20. Salay, R., Zschaler, S., Chechik, M.: Correct reuse of transformations is hard to guarantee. In: ICMT. Volume 9765 of LNCS., Springer (2016) 107–122
21. Wuliang, S., Combemale, B., Derrien, S., France, R.: Using model types to support contract-aware model substitutability. In: ECMFA. Volume 7949 of LNCS., Springer (2013) 118–133
22. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Melange: A meta-language for modular and reusable development of DSLs. In: SLE, ACM (2015) 25–36
23. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Trans. Model. Comput. Simul. **12**(4) (2002) 290–321
24. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G.M., Syriani, E., Wimmer, M.: Model transformation intents and their properties. SoSyM **15**(3) (2014) 685–705
25. Etien, A., Muller, A., Legrand, T., Paige, R.F.: Localized model transformations for building large-scale transformations. SoSyM **14**(3) (2015) 1189–1213
26. Morin, B., Klein, J., Kienzle, J., Jézéquel, J.M.: Flexible model element introduction policies for aspect-oriented modeling. Volume 6395 of LNCS., Springer (2010) 63 – 77
27. Cuadrado, J.S., García Molina, J.: Approaches for model transformation reuse: Factorization and composition. In: ICMT. LNCS, Springer (2008) 168–182
28. Kleppe, A.: MCC: A model transformation environment. In: ECMDA-FA. Volume 4066 of LNCS., Springer (2006) 173–187
29. Sutîi, A., van den Brand, M., Verhoeff, T.: Exploration of modularity and reusability of domain-specific languages: an expression DSL in metamod. Computer Languages, Systems & Structures **51** (2018) 48–70
30. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: Variability-based model transformation: formal foundation and application. Formal Asp. Comput. **30**(1) (2018) 133–162
31. Wagelaar, D., Straeten, R.V.D., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. SoSyM **9**(3) (2010) 285–309
32. Wimmer, M., et al.: Surveying rule inheritance in model-to-model transformation languages. JOT **11**(2) (2012) 3: 1–46
33. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3) (2006) 621–645
34. Kahani, N., Bagherzadeh, M., R. Cordy, J., Dingel, J., Varro, D.: Survey and classification of model transformation tools. SoSyM **In press** (2018)
35. Mengerink, J., Serebrenik, A., Schiffelers, R.R.H., van den Brand, M.G.J.: Automated analyses of model-driven artifacts: obtaining insights into industrial application of MDE. In: IWSM-Mensura, ACM (2017) 116–121