HAL
open science

Comparing and Classifying Model Transformation Reuse Approaches across Metamodels

Jean-Michel Bruel, Benoit Combemale, Esther M Guerra, Jean-Marc Jézéquel, Jörg Kienzle, Juan de Lara, Gunter Mussbacher, Eugene Syriani, Hans Vangheluwe

To cite this version:

Jean-Michel Bruel, Benoit Combemale, Esther M Guerra, Jean-Marc Jézéquel, Jörg Kienzle, et al.. Comparing and Classifying Model Transformation Reuse Approaches across Metamodels. Software and Systems Modeling, 2020, 19 (2), pp.441-465. 10.1007/s10270-019-00762-9 . hal-02317864

HAL Id: hal-02317864

https://inria.hal.science/hal-02317864

Submitted on 16 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SoSyM manuscript No.
(will be inserted by the editor)

Comparing and Classifying Model Transformation Reuse Approaches across Metamodels

Jean-Michel Bruel · Benoit Combemale · Esther Guerra · Jean-Marc Jézéquel · Jörg Kienzle · Juan de Lara · Gunter Mussbacher · Eugene Syriani · Hans Vangheluwe

the date of receipt and acceptance should be inserted later

Abstract Model transformations are essential elements of Model-driven Engineering (MDE) solutions, as they enable the automatic manipulation of models. MDE promotes the creation of domain-specific metamodels, but without proper reuse mechanisms, model transformations need to be developed from scratch for each new metamodel.

In this paper, our goal is to understand whether transformation reuse across metamodels is needed by the community, evaluate its current state, identify practical needs and propose promising lines for further research. For this purpose, we first report on a survey to understand the reuse approaches used currently in practice and the needs of the community. Then, we propose a classification of reuse techniques based on a feature model, and compare a sample of specific approaches – model types, concepts, a-posteriori typing, multilevel modeling, typing requirement models, facet-oriented modeling, mapping operators, constraint-based model types, and design patterns for model transformations – based on this feature model and a common example. We discuss strengths and weaknesses of each approach, provide a reading grid used to compare their features, compare with community needs, identify gaps in current transformation reuse approaches in relation to these needs and propose future research directions.

Keywords Model Transformation, Reuse, Survey, Classification, Feature Model

J-M. Bruel
University of Toulouse, IRIT, France E-mail: bruel@irit.fr

B. Combemale
University of Toulouse, IRIT, France
E-mail: benoit.combemale@irit.fr

E. Guerra
Universidad Autónoma de Madrid, Spain
E-mail: Esther.Guerra@uam.es

J-M. Jézéquel
University of Rennes, Inria, CNRS, IRISA, France
E-mail: jean-marc.jezequel@irisa.fr

J. Kienzle
McGill University, Canada E-mail: Joerg.Kienzle@mcgill.ca

J. de Lara
Universidad Autónoma de Madrid, Spain
E-mail: Juan.deLara@uam.es

G. Mussbacher
McGill University, Canada E-mail: gunter.mussbacher@mcgill.ca

E. Syriani
Université de Montréal, Canada E-mail: syriani@iro.umontreal.ca

H. Vangheluwe
University of Antwerp, Belgium and McGill University, Canada
E-mail: Hans.Vangheluwe@uantwerpen.be

1 Introduction

Model-driven Engineering (MDE) is being used for engineering evermore numerous and complex systems [73]. Recent studies evince its increasing adoption by industry, but also report on the scalability issues that this entails in terms of size, variety, complexity and maintenance of the artefacts involved in MDE-based solutions [5,27,48,51]. In this scenario, model transformations (MTs) are also becoming more and more complex pieces of software. Hence, like for any other type of software [31], *reuse mechanisms* for MTs have been proposed to avoid reimplementing a transformation from scratch every time a new but related need arises [32].

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. On the one hand, many use cases of MT reuse have been identified in the literature [32], providing useful classifications. On the other, several approaches to reuse across metamodels have been proposed by different researchers [15,21,36,38,41,44,45,62,65,71,74,77]. Since

the use cases of MT reuse imply very different trade-offs among properties such as type-safety, performance, expressiveness and user-friendliness, no single MT reuse approach fits them all.

In this work, we start by comparing practical MT needs compiled from the authors' experience with research approaches to MT reuse, analysing the latter to check whether they cover needs indicated by the community. In particular, we want to understand whether MT reuse across metamodels is a real recurring need faced by developers, identify how they currently solve these reuse scenarios, and what would be their requirements for an ideal MT reuse mechanism. For this purpose, we surveyed over a hundred researchers and users of transformation languages. Overall, we found that most of them needed to reuse MTs, most used ad-hoc copy-paste techniques, and the vast majority mentioned that dedicated reuse techniques would be highly desirable.

We then looked at the state of the research in MT reuse. For this purpose, methodologically, we propose a novel classification of MT reuse approaches that work across metamodels, and compare a sample of nine specific approaches — namely model types [21,65], concepts [36,62], a-posteriori typing [38], multilevel modeling [44], transformation typing requirements models (TRMs) [42,45], facet-oriented modeling [41], constraint-based model types [77], mapping operators (MOps) [74], and design patterns for MTs [15] — with the help of a feature model developed for this aim, and a common example. We discuss strengths and weaknesses of each proposal, provide a reading grid to compare their features, compare with community needs and requirements, and identify gaps in current reuse approaches with respect to these needs.

We presented an earlier version of this work at ICMT 2018 [9]. The present paper extends the former by reporting on the results of a survey collecting the requirements and needs of MDE researchers and transformation language users regarding MT reusability. We have included a comparison of four additional reuse approaches: typing requirements models, facet-oriented modeling, constraint-based model types, and mapping operators. Finally, we have included guidelines, in the form of a decision tree, to help developers in choosing a reuse approach.

The paper is organized as follows. Section 2 sets the stage for reuse mechanisms across metamodels, presenting a running example, and motivating the need of such mechanisms by reporting on a survey about transformation reuse. Section 3 defines classification criteria for reuse mechanisms using a feature model. Section 4 compares nine existing approaches based on the classification and the running example. Section 5 discusses trade-offs and coverage of needs identified in the survey, and presents guidelines for selecting a transformation reuse approach. Section 6 overviews related classification attempts and reuse techniques, and Section 7 concludes by identifying challenges for the MT community. The feature model, the configurations for each approach, and the anonymized raw results of the survey are available at `http://bit.ly/bellairs18`.

2 Setting the stage for reuse: Why is it needed?

In this section, we motivate the need for model transformation reuse with a running example (Section 2.1) and briefly report on a motivation survey we conducted (Section 2.2).

2.1 Motivating example

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 [64]. 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 any general-purpose language (GPL) like Java, or a domain-specific language (DSL) like ATL [24], ETL [30], or Kermeta [23], but to stay language-agnostic, we only show a postcondition 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 this kind of languages, which would require a flattening operation to obtain all states of a state machine, all packages of a project, or all subgoals of a given goal.

Without proper reuse mechanisms, a flattening transformation 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 writing an adapter transformation which translates the models of interest to the metamodel accepted by the reused MT. Alternatively, polymorphic or parametric reuse mechanisms provided by various GPLs can also be used (e.g., generics and subtyping in most object-oriented languages), but suffer from important limitations (e.g., type

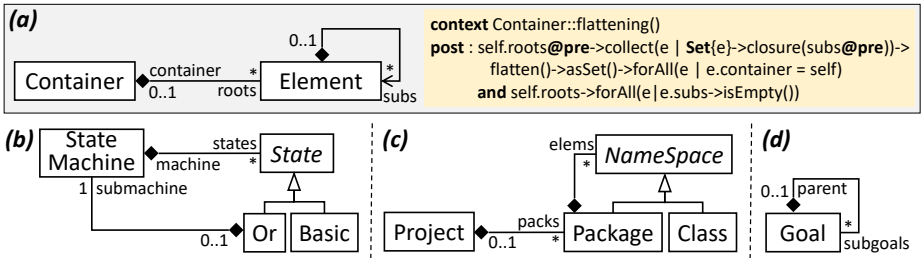


Fig. 1: (a) Reusable model transformation scheme. (b, c, d) Metamodels for which the model transformation wants to be reused.

group subtyping [13]) or accidental complexity (e.g., generics of generics for as many concepts as provided by the metamodel) that prevent any industrial use cases.

None of the previous approaches are optimal. In the first case, manual adaptation is time-consuming, error-prone, hardly scalable, and leads to well-known maintenance problems with code clones [25]. 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 bottom-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 M' 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 extra transformation in addition to the reused one. It also complicates traceability, and moreover, a reverse MT may be needed to transform the result back to the original model's metamodel MM'.

At this point, our question is whether reuse across metamodels is common at all in practice. We would like to understand how it is currently solved by MT users, and possible requirements for ideal reuse approaches. It must be noted that the MT research community has proposed several approaches to facilitate reuse across metamodels, like model typing, a-posteriori typing, concepts, multilevel modeling and transformation patterns, among others [41,45,71, 74,77]. These approaches have different trade-offs and are applicable in different scenarios and contexts. Hence, there is a need to classify and compare them to know which approach to use in a given situation, and to check whether their capabilities fulfill requirements from MT users.

2.2 A need from the community

We polled a sample of the modeling community to verify if there is a real need for MT reuse. The 114 respondents were participants of the MODELS 2018 conference and members of MDE mailing lists who voluntarily answered our questionnaire. They are all experienced (academic and industrial researchers) in MT development. The details of the survey and result data can be found online [1].

Reuse context. Most participants in the survey worked in academia (72%), followed by industry (17%) and research centers (11%). Hence, the survey is biased towards academic researchers, but this is still valid for our goal of understanding whether MT reuse is perceived as a need. Expanding the survey towards industry is future work.

To understand the reuse context, we asked the participants for the languages they used to write MTs. They were free to include any number of MT or general-purpose languages, as this was an open question. Fig. 3 shows the distribution of the most popular languages mentioned (those with at least 3% support) distinguishing also between their usage in academia, industry and research centers. ATL [24] appears as the most used one, followed by code generator languages (Acceleo is the most used at 6%), GPLs (Java is the most used at 14%), QVT [19] (QVTr being the most used), Epsilon [55], Kermeta [22] and others. The most popular language among the participants from both academia (49%) and research centers (62%) is ATL, in the latter case tied with code generator languages (62%). In industry, we found that GPLs are the most widespread option (58%), followed by ATL (26%) and code generator languages (26%).

Reuse need and current reuse techniques. A majority of the participants (60%) stated that they encountered the need to reuse MTs in their practice. A large part of the participants reported to use the basic technique of copy-paste (46%). Other applied techniques include the implementation of adapters (20%), early modularization (11%), and higher-order transformations (8%). Other specific techniques were also mentioned, but with a representativity lower than 5%, such as cloning [69], subtyping [8,21,76], localized transformations [16] and annotations [10].

Desired reuse features. Basic reuse techniques, such as copy-paste, fail to scale at the level of complexity found in

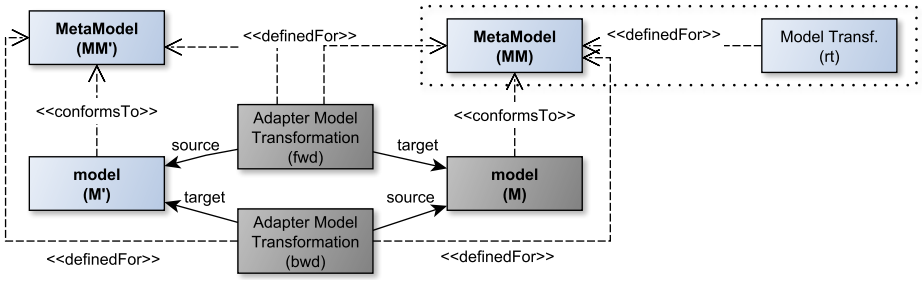


Fig. 2: Explicit model adaptation approach to MT reuse

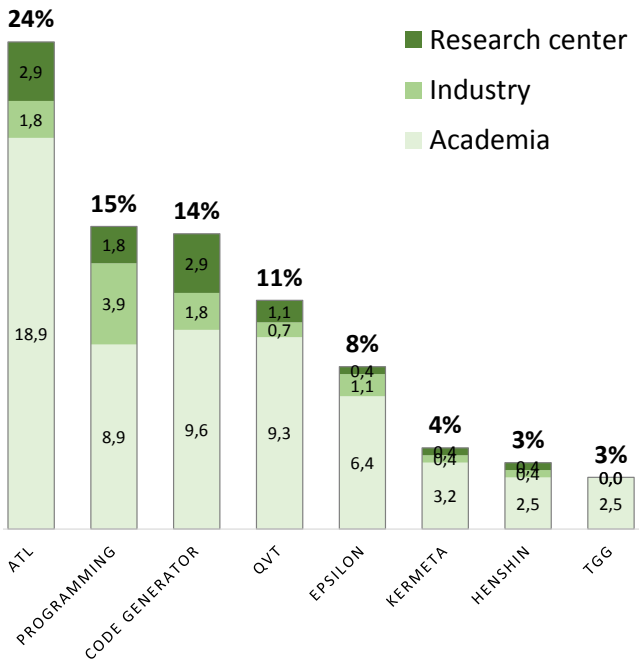


Fig. 3: Distribution of most popular MT languages used

practice. Hence, the need for advanced reuse mechanisms is broadly accepted by the participants of the survey at 87%.

The features that the participants identify from a list of given options as the most relevant for a MT reuse approach to be useful in practice include enabling partial reuse (22%), support for analysis of correct reuse (17%), being declarative (16%), being MT language-independent (14%), support for discovery of reusable MTs (14%), and being black box (13%). Interestingly, the execution cost for running a reused transformation is of low interest (<5%). Further desirable features proposed by the participants were MT parametrization with metamodels (2%), and systematic modularization (2%).

Survey conclusions. Almost 90% of the survey respondents clearly stated that there is a need to reuse MTs across metamodels. They have been reusing transformations in various scenarios, such as cross-platform development, metamodel evolution, and software product family development. However, they typically reuse in an ad-hoc way through copy-paste or by writing adapter transformations each time. Although they strongly believe that dedicated mechanisms to reuse are needed, they are not familiar with such mechanisms. Nevertheless, they are eager to find MT reuse approaches that offer a number of features, such as language independence and partial reuse. Therefore, the remainder of this article helps shedding the light on existing approaches that may be beneficial to the MT community.

3 Classification of MT reuse approaches

Given the analysis of the practical needs raised by the community, next we analyse the design space of MT reuse approaches. For this purpose, we introduce a feature model [28] to classify the alternatives for MT reuse across metamodels. The model, shown in Figs. 4 and 5, presents the features of the reuse mechanism as well as properties of the reused transformation. The features appear equally numbered in the figures and in the following paragraphs to facilitate readability.

Overall, the feature model contains 70 features, enabling more than 510 000 different configurations. The model was elicited during the 1st Workshop on Unifying Software Reuse, a one-week intensive research workshop at Bellairs with participation of reuse experts for MDE (`http://www.bellairs2018.ece.mcgill.ca/`). In the following definitions of features, rt denotes the MT to be reused.

(1) 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 [62], as a pattern [15], or the metamodel the MT is defined on may need to be sliced [63]. All other kinds of reuse are considered *opportunistic*.

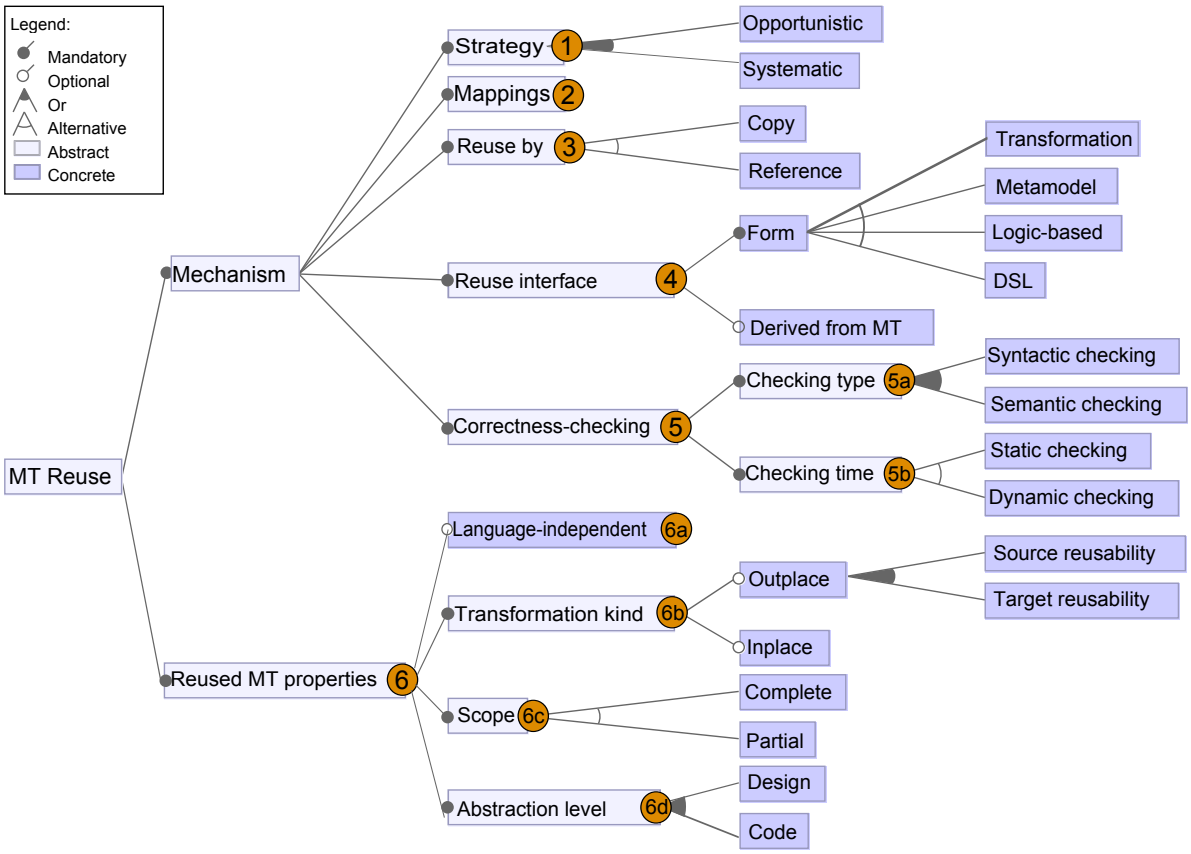


Fig. 4: Feature model: mechanisms for reuse and scenarios of reuse (the *Mapping* feature is expanded in Fig. 5)

(2) Mappings. A reusable transformation rt, defined over a metamodel MM, is applicable to a number of different metamodels MM'. Typically, to enable reuse, some kind of mapping needs to be established between the elements of MM and MM'. The way to specify this mapping or correspondence depends on the reuse approach, and determines the set of metamodels where rt can be reused. Fig. 5 shows the design space for mapping specifications, which comprises the following features:

- ***(2a) 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*: several elements of MM can be mapped to the same element in MM'. Finally, the most general kind of mapping is *many-to-many*: elements in both MM and MM' can be mapped several times.
- ***(2b) 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*).
- ***(2c) 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 with respect to a metamodel).
- ***(2d) 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 [65] or structural similarity criteria [45, 74].
- ***(2e) 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 – except the mapping operators [74] – only support one application context at a time.
- ***(2f) Adaptation.*** To widen the number of metamodels where a transformation can be reused, several ap-

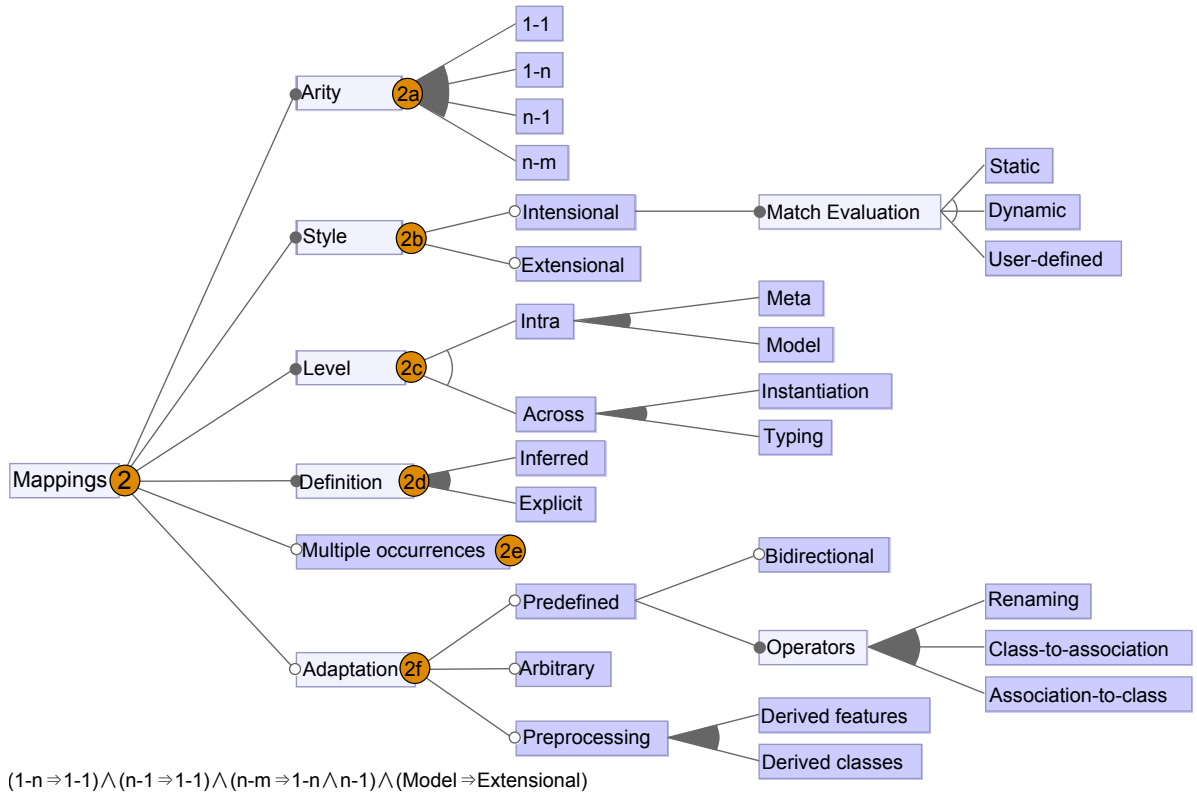


Fig. 5: Feature model: specification of mappings

proaches provide mechanisms to bridge structural 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* [62, 74] (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 [14,62]. For example, if an association needs to be mapped to a class, we may create a virtual class to facilitate the mapping. However, this may require a preprocessing step to populate the model with a derived object for each association instance.

(3) Reuse by. This feature refers to whether the original transformation is *copied* or *referenced*. In the *copy-paste* approach, also known as *clone-and-own* (cf. Section 2.1), the developer reuses a modified copy of rt to perform 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.

(4) 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 necessary classes and features in the context of reuse [15,21,62, 65], a *logic-based* specification stating the constraints that a metamodel should fulfill to ensure a correct MT reuse [77], or a model describing metamodel requirements using a domain-specific language (*DSL*) [42,45]. Sometimes, this reuse interface can be (semi-)automatically *derived from the MT* [42,45,63,77].

While the above-mentioned interface kinds yield a black-box approach to reuse, the interface for reuse in white-box approaches is the reusable *transformation* or an abstraction of it [15,74]. This is appropriate when a larger MT is to be composed out of smaller fragments. Both interface kinds can be combined.

(5) Correctness checking. Different approaches make different choices on how and when the correctness of rt with respect to m' and MM' should be checked:

– ***(5a) Checking type.*** Checking can be either *syntactic*, e.g., simple type checking, or *semantic*, typically

Table 1: Classification of MT reuse approaches

Feature	Model-Typing	Concepts	A-posteriori	Facets	Multilevel	MT Patterns	MOps	Typing Reqs	Const. Types
Mechanism									
Strategy	Systematic Opportunistic	Systematic Opportunistic	Systematic Opportunistic	Systematic Opportunistic	Systematic Opportunistic	Systematic	Systematic	Opportunistic	Opportunistic
Reuse by	Reference	Copy	Reference	Reference	Reference	Copy	Reference	Reference	Reference
Reuse interface	Metamodel Can be derived	Metamodel Can be derived	Metamodel	Metamodel	Metamodel	Transformation	Transformation	DSL Derived	Logic-based Derived
Checking type	Syntactic Semantic (pre, post)	Syntactic	Syntactic	Syntactic	Syntactic	Syntactic	Syntactic	Syntactic	Syntactic
Checking time	Static (type-level) Dynamic (inst-level)	Static	Static (type-level) Dynamic (inst-level)	Dynamic	Static	Static	Static	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, n-1, n-m	1-1, 1-n	1-1	1-1, 1-n, n-1, n-m[a]	1-1, n-1	1-1, n-1
Style	Extensional	Extensional	Extens. (type-level) Intens. (inst-level) Dynamic match	Extensional Intensional Dynamic match	Extensional	Extensional	Extensional	Extensional	Extensional
Level	Intra/meta	Intra/meta	Intra/meta	Intra/meta	Across/ instantiation	Across/ typing	Intra/meta	Intra/meta	Intra/meta
Definition	Inferred	Explicit	Explicit	Explicit	Explicit	Explicit	Explicit	Inferred	Inferred
Multiple occ.	no	no	no	no	no	no	yes	no	no
Adaptation	Renaming, derived feats	Renaming, c-to-a, a-to-c, arbitrary, derived feats, derived classes	Renaming, arbitrary, bidirectional, derived feats	Renaming, arbitrary, bidirectional, derived feats	Renaming, derived feats	Renaming	Arbitrary[b]	Renaming[c]	–
Reused MT properties									
Lang. indep.	yes	no	yes	yes	yes[d]	yes[e]	yes[e]	no	no
Transf. kind	Inplace Outplace (M2M & M2T)	Inplace [36] Outplace [62] src/tar reuse	Inplace Outplace src/tar reuse	Inplace Outplace src/tar reuse	Inplace Outplace src/tar reuse	Inplace Outplace src/tar reuse	Outplace src/tar reuse	Inplace Outplace src/tar reuse	Inplace
Scope	Complete	Complete	Complete	Complete	Partial[f]	Partial	Complete	Complete	Complete
Abstrac. level	Code	Code	Code	Code	Code	Design	Design	Code	Code

[a] Preprocessing of derived features for alignment
[b] With conditional MOps using OCL for filtering
[c] When class names are unimportant (anonymous classes)
[d] But for better usability, transformation languages need to be multilevel aware
[e] By additional code generators
[f] Through refining transformations [36]

also verifying the satisfaction of well-formedness rules expressed in OCL, or additional semantic conditions capturing the transformation intent (e.g., bisimulation relations) [59].

– ***(5b) 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.

(6) Properties of reused transformation. Some reuse approaches may be limited to a particular transformation language, kind of transformation (inplace, outplace), application scope (whole transformations or parts) or abstraction level (transformation design or code). These alternatives give raise to the following features of the feature model:

– ***(6a) Language independence.*** 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 [62] or graph transformation [37,47,71]).

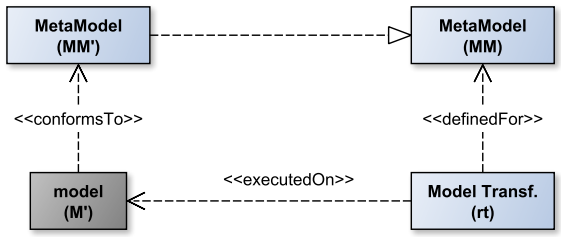
– ***(6b) Transformation kind.*** The reused transformation can be either *inplace* or *outplace* (i.e., model-to-model). In the former case, the mechanism needs to ensure that both read and write accesses to the model are correct. In the latter case, the new context of reuse can be for the source metamodel, which is typically read-only (*source reusability*), for the target metamodel, which is typically write-only (*target reusability*), or for both.

– ***(6c) Scope.*** The reused artifact can be a *complete* model transformation or a part of it, e.g., a rule (*partial*).

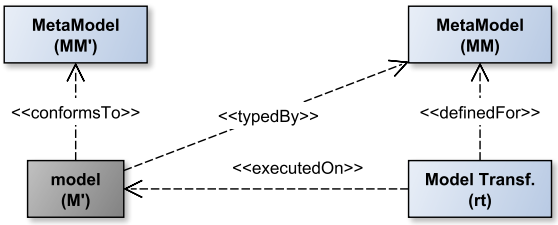
– ***(6d) Abstraction level.*** Reuse can be at the *design* level, e.g., in the form of design patterns [15], or directly at the implementation level to reuse transformation *code*.

The previous features are orthogonal but for the constraints implied by the feature model and a few dependencies that concern the mapping specifications. Such dependencies are expressed as a logical formula in Fig. 5. Specifically, the ability to specify one-to-many and many-to-one mappings imply the ability to specify one-to-one mappings; many-to-many mappings imply the ability to specify one-to-many and many-to-one mappings; and supporting intra-level mappings (i.e., mappings between artifacts at the same metalevel) implies the need to have an extensional mapping style.

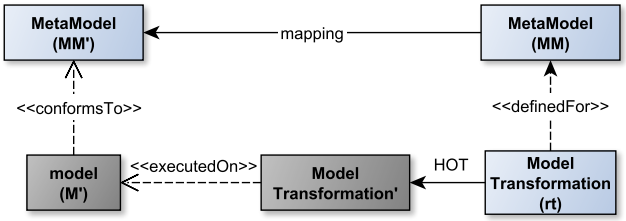
While we do not provide a formal proof of the correctness or completeness of the feature model, the next section



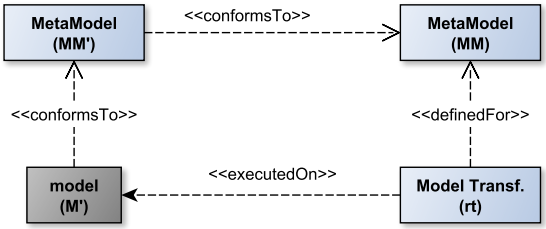
(a) Subtyping (model typing [21,65])



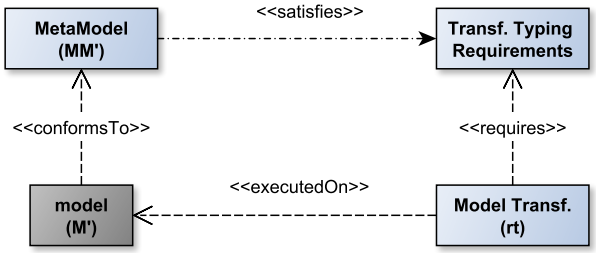
(b) Retyping (a-posteriori typing [38], facets [41])



(c) High-order transformation/generative (concepts [62], MT patterns [15], MOps [74])



(d) Instantiation (multilevel modeling [44])



(e) Transformation typing (TRMs [45], constraint model types [77])

Fig. 6: Different techniques enabling MT reuse across metamodels, and references to approaches using them

shows its instantiation for many of the currently existing MT reuse approaches across metamodels. This provides some confidence in this respect.

4 Comparison of some existing approaches

In this section, we analyse nine prominent MT reuse approaches: model typing, concepts, a-posteriori typing, facets, multilevel modeling, design patterns for MTs, MOps, typing requirements models, and constraint-based model types. Table 1 summarizes how each of them instantiates the feature model introduced in the previous section (cf. Figs. 4 and 5). In the table, we can see that the considered approaches cover many of the features in the feature model, including both opportunistic and systematic reuse, reuse by copy and by reference, several reuse interface kinds, syntactic and semantics checks, static and dynamic checks, different mapping styles, etc. While in this section, we focus on the presentation of the approaches, Section 5.1 will provide an in-depth analysis of this table leading to the identification of gaps of the reuse space by existing MT reuse approaches.

Fig. 6 shows a scheme of the techniques used by the MT reuse approaches presented in the remainder of this section. Model typing is based on establishing a subtyping relation between metamodels (schema shown in Fig. 6a). A-posteriori typing and facet-oriented modeling work by retyping the model so that the reused MT can be applied to it (Fig. 6b). Concepts rely on genericity to rewrite the MT using a high-order transformation to make it applicable to a particular metamodel (Fig. 6c). MT patterns and MOps use a generative approach to synthesize MT code from a design pattern or a (composite) mapping operator. The reuse granularity in these two latter techniques is fine-grain, as they are based on libraries of generic operators and patterns that are to be used in combination. Multilevel modeling exploits the typing relation to apply the MT two (or more) metalevels below (Fig. 6d). Instead of using a metamodel as the interface to reuse, typing requirements models and constraint-based model types extract from the transformation the typing requirements needed by other metamodels to ensure a correct reuse (Fig. 6e). These requirements are expressed either with logics [77] or a DSL [45], and checked to assess whether a given metamodel satisfies them.

In the following, we provide more details on their working scheme using the running example, as well as their support by tools.

4.1 Model typing

Working scheme. Model Types were introduced by Steel et al. [65] 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 [12,21,76]. In practice, subtyping relationships are graph isomorphisms (total or partial, possibly with adapters) that ensure any model transformations defined on top of a super type to be safely applied to any model typed by a subtype.

Running example. Fig. 7 illustrates model typing, showing how to reuse the flattening MT defined on model type MTy 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').

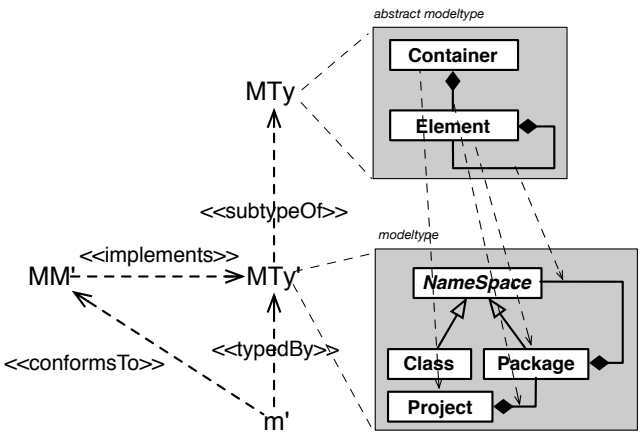


Fig. 7: Reuse with model typing

Tool support. The supporting tool Melange[1] employs adapter generators at compile time to ensure the adaptation of the instances of the targeted metamodel at runtime, when the MT is actually applied [12].

[1] http://melange.inria.fr/

4.2 Concepts

Working scheme. Inspired by generic programming [18], concepts were proposed in [36] as a mechanism to express requirements for generic model management operations and transformations. In this context, 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 (also called *templates*) 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 [62] enable more flexible bindings by the use of OCL expressions in mappings, which get injected in the rewritten MT code.

Fig. 8 illustrates the working scheme of concepts to reuse a model-to-model transformation template written in ATL. The transformation is typed by one source concept (from) and one target concept (to). Any of them can be bound to a specific metamodel (in the figure, only the source concept is bound). The binding consists of class mappings (e.g., class A to class P), feature mappings, and adapter expressions (e.g., A.num to P.name.length()). The figure shows the result of the template instantiation, which yields a new transformation typed over the metamodels the concepts were bound to.

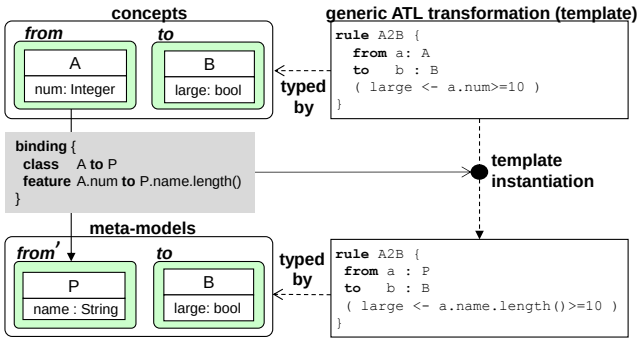


Fig. 8: Binding of concept to metamodel, and MT adaptation (sketch)

Running example. Fig. 9 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 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.

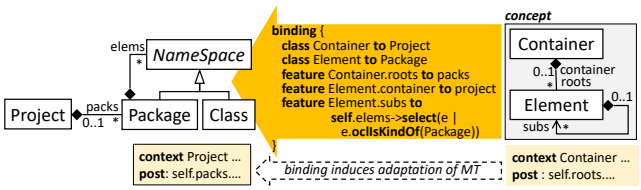


Fig. 9: Using concepts to solve the running example (sketch)

Tool support. Concepts are supported by the METADEPTH tool[2], and are applicable to any model management language of the Epsilon family [55]. A specific implementation for ATL, with an expressive binding DSL [62,63] and facilities to refactor a MT to obtain a simpler concept, is supported by the bentō tool[3].

4.3 A-posteriori typing

Working scheme. A-posteriori typing [38] 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. 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.

Figs. 6b and 10 show the working scheme of this approach: a model typed by an arbitrary metamodel is assigned new types from the metamodel a MT is defined on, and as a result, the MT can be executed as-is on the model. In Fig. 10, we indicate the new types as stereotypes (e.g., <<A>> and <<num>>).

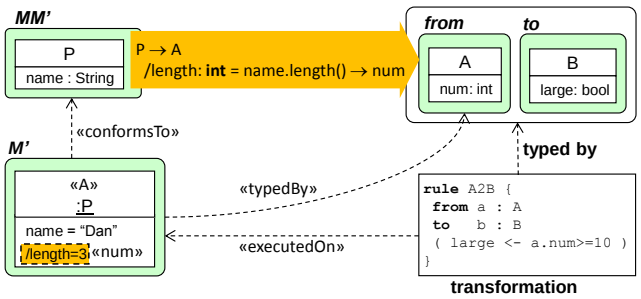


Fig. 10: Type-level a-posteriori typing

[2] http://metadepth.org
[3] http://sanchezcuadrado.es/projects/bento/

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 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.

As a difference with concepts, a-posteriori typing creates a new, derived typing, and hence the original transformation does not need to be rewritten. This fact allows expressing some kinds of transformations as a-posteriori specifications, as Fig. 11 illustrates. In such a case, the result of the transformation would be obtained by slicing the model to retain only the elements typed by the target metamodel (metamodel to in the figure). As the new typing is updated whenever the model changes, one obtains incrementality for free. However, a-posteriori typing specifications are less expressive than full-fledged MT languages (see [38] for details).

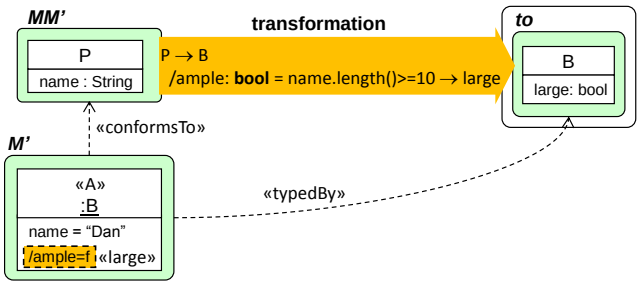


Fig. 11: Expressing a transformation as a retyping

Running example. Fig. 12 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 and facet-oriented modeling.

Tool support. A-posteriori typing is supported by the tool METADEPTH, where it can be used with any model management language of the Epsilon family.

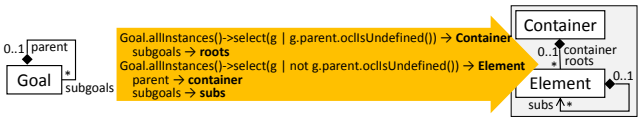


Fig. 12: A-posteriori instance-level specification for the flattening of goal models

4.4 Facets

Working scheme. Inspired by role-based modeling [66], facet-oriented modeling [41] permits adding slots, constraints and types (i.e., facets) to objects dynamically. Hence, an existing object can be added (and removed) facets, whose definition is taken from an existing metamodel. The new slots brought by the facet become transparently accessible from the *host* object of the facet. The conditions for a host object to take or drop some facet are specified using so-called facet laws.

This approach enables MT reuse by considering the metamodel over which the reusable MT is defined as the facet metamodel, and specifying facet laws that assign suitable facets to the model objects and features the transformation is to be reused on. Facets are similar to a-posteriori typing, but facet slots do not need to be backed by existing object slots or be derived, as required by a-posteriori typing. For example, in Fig. 10, a-posteriori typing requires length to be a derived attribute, and hence read-only. Instead, with facets, the objects of type P can be increased with an extra mutable attribute num, which is initialized according to the given expression, and can be changed later (see Fig. 13). Objects can also share facets, hence being a way to synchronize attribute values among sets of objects.

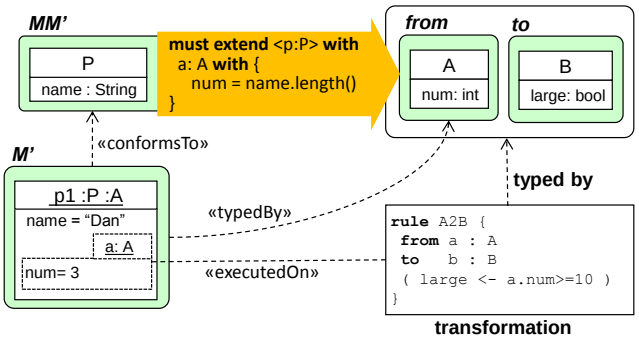


Fig. 13: Working scheme of facets

Facet laws support two matching styles: intentional, which is similar to transformation rules (see for example Fig. 13), and extensional, by selecting specific objects based on their identifiers.

Running example. Fig. 14 shows the facet laws needed to assign appropriate facets to goal models, so that the flattening transformation becomes reusable as-is on them.

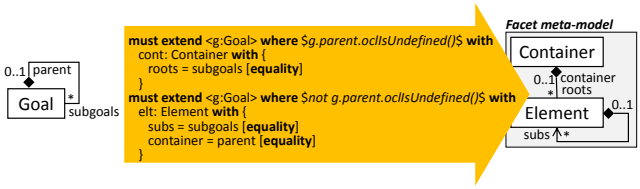


Fig. 14: Facet laws for reusing the flattening operation over goal models

Tool support. Facet-oriented modeling is supported by an experimental version of the METADEPTH tool[4].

4.5 Multilevel modeling

Working scheme. Multilevel modeling [4,43] provides a way to enhance flexibility in modeling by enabling an arbitrary number of metalevels, where elements may be both types and instances at the same time. In particular, elements are instances with respect to the metalevel above, and types with respect to the metalevel below. For this reason, they are uniformly called clabjects (from the contraction of the words *class* and *object*). This approach facilitates the definition of domain-specific metamodeling languages and families of languages [44], 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. 15 shows an example of multilevel model, where a model MM is instantiated into the model MM', and this one is instantiated into the model M'. Models and their elements can declare a *potency* to control their instantiation depth, or otherwise, they receive the potency from their container element. In the example, MM has potency 2 (indicated after the "@" symbol), meaning that it can be instantiated at two consecutive metalevels below, while A takes potency 2 as it is contained in MM. At every metalevel, the potency of an element is one less than the potency of its type. Clabjects with potency 0 (like p1) cannot be instantiated, and attributes can only receive a value when they have potency 0 (like p1.num). When used for MT reuse, multilevel modeling can emulate attribute bindings using OCL constraints. For example, in Fig. 15, the invariant numBinding attached to clabject P permits deriving a value for attribute num, which

[4] http://metadepth.org/mtl

can then be used in the transformation defined two levels above.

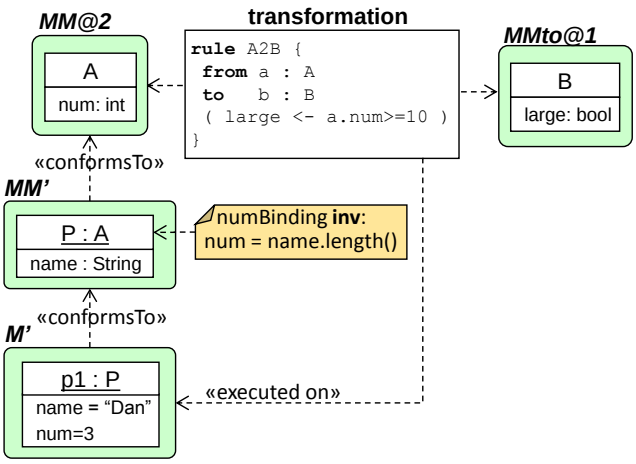


Fig. 15: Working scheme of multilevel modeling reuse

Running example. Fig. 16 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.

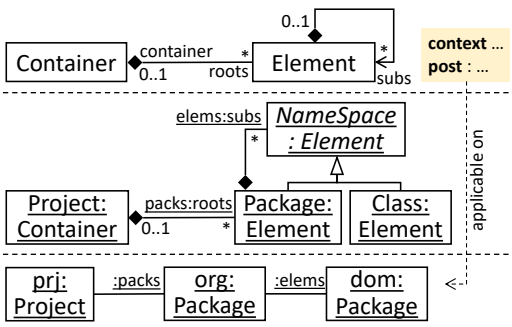


Fig. 16: Reuse by multilevel modeling

Tool support. Several tools support transformation reuse via multilevel modeling. Multilevel transformations were originally proposed in [44], and are supported within the METADEPTH tool for the Epsilon languages. Melanee [3] is a multilevel modeling tool based on Eclipse/EMF, which supports defining multilevel transformations using ATL. MultEcore [50] is a recent multilevel modeling tool with a dedicated language to express model transformations.

4.6 Design patterns for model transformations

Working scheme. 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 [15] 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.

Running example. The top of Fig. 17 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. 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) and removes the subs links. The leaves rule creates a roots relation from the container to all elements (if it does not yet exist), while removing the trace links.

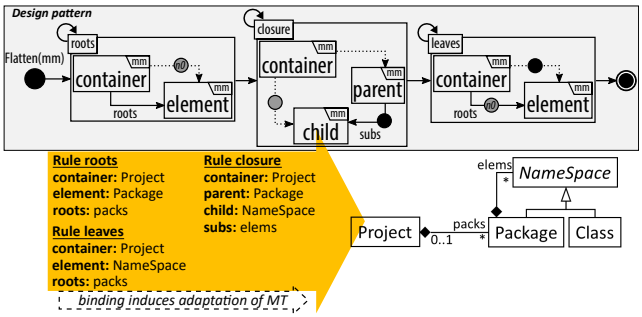


Fig. 17: Binding of flattening design pattern to metamodel

The Flatten design pattern and the mapping in Fig. 17 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. 6c for the MT patterns approach, MM corresponds to the metamodel of DelTa (see [15]), 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'.

Tool support. A prototypical tool called DelTaEMF[5] is available to generate instances of a design pattern into various MT languages, by interactively binding pattern participants to metamodel elements.

4.7 Composite mapping operators

Working scheme. In [74], the authors propose a library of reusable, generic mapping operators (MOps) to adapt one metamodel MM' to another metamodel MM. Each MOp is generic and metamodel-independent. This way, similar to transformation patterns, this approach provides a generic library of reusable operators which can be selected and composed to build a transformation (i.e., the operators are the artifacts to be reused). The mapping is enacted by generating an adapter transformation from MM' to MM using a concrete MT language.

The authors distinguish between kernel and composite MOps. For instance, a *copier* is a composite operator that creates one target object per source object, and is composed of kernel MOps of the following three kinds: *C2C* to create classes, *A2A* to create attributes, and *R2R* to create references. Another example of composite MOp is a *horizontal partitioner* that splits the source object set into several target classes by means of a condition.

Running example. Fig. 18 shows how to use MOps to translate goal models into container models over which the flattening operation can be applied, therefore following the explicit adaptation approach to MT reuse illustrated in Fig. 2. Specifically, three *conditional copier* operators split the Goal class into the two target classes Container and Element depending on the value of its reference parent. From this specification, an adapter transformation from goal models to container models is generated.

Just like in the case of a-posteriori typing (cf. Fig. 11) implementing reusable MTs using MOps is also possible. Such MTs would be reused by mapping the operator ports to the elements of other metamodels, similarly to bindings in concepts. However, this is only possible for outplace transformations, but not for inplace transformations like the flattening operation.

Tool support. While tool support for MOps was available in previous Eclipse versions, the approach is currently unsupported.

4.8 Typing requirements models

Working scheme. A domain requirements model (DRM) [42,45] describes the minimal requirements

[5] https://github.com/hergin/DelTaEMF

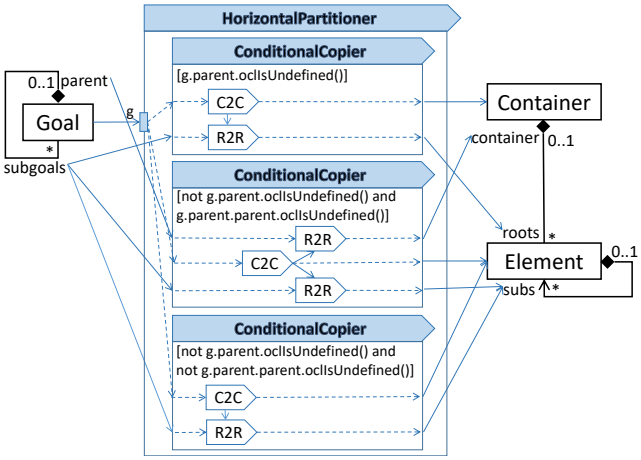


Fig. 18: Reuse by MOps

that an inplace MT imposes on a metamodel to obtain a correctly typed transformation. This way, the transformation can be reused as-is with all metamodels conforming to the DRM. The approach is applicable to model-to-model transformations as well, in which case, the requirements are expressed by means of a typing requirements model (TRM) consisting of the DRMs of the source and target metamodels of the transformation, together with a feature model expressing dependencies between them. In [42, 45], a process to automatically extract the TRM of ATL transformations is proposed. This permits reusing an ATL transformation as-is for any pair of metamodels that satisfy the TRM, including the metamodels over which the transformation was originally defined for.

Running example. Fig. 19 shows on top the DRM expressing the metamodel typing requirements for the flattening operation. The DRM requires the metamodel to contain a class named Container, as well as two classes with any name. The latter two *anonymous* classes could be matched by the same or different classes in concrete metamodels, or even be the class Container provided this declares the necessary fields. In particular, class Container should define a multi-valued reference roots whose type and minimum cardinality can be any (we use "?" to indicate that the minimum cardinality is open); the target class of reference roots should define an optional reference called subs, whose type can be any; and the target class of reference subs should define a mono-valued reference called container with type Container. The three classes can be either abstract or concrete (indicated by the encircled "AC"), and their requirements may be fulfilled by them or by all its subclasses (indicated by the encircled white triangle).

The lower part of the figure shows three metamodels that conform to the DRM, meaning that the transformation can be reused with them. In the left metamodel, both unnamed

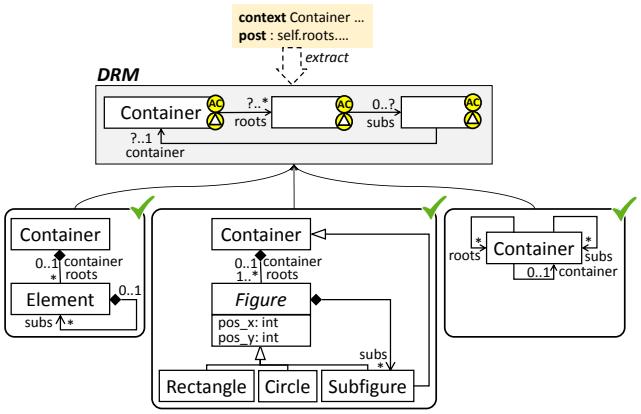


Fig. 19: Domain requirements model, and some conforming metamodels

classes in the DRM are matched by the same class Element, as it declares the necessary references subs and container. In the metamodel in the middle, the unnamed classes are matched by two different classes: Figure (which is abstract) and Subfigure (which is concrete). In the metamodel to the right, all three classes in the DRM are matched by a single class: Container. Interestingly, this approach does not require establishing an explicit binding, but a satisfaction relation is checked.

Tool support. This approach is supported by the TOTEM tool[6], an Eclipse plugin able to extract TRMs from ATL transformations, without requiring the transformation metamodels.

4.9 Constraint-based model types

Working scheme. Similar to the previous approach, constraint-based model types express typing requirements for model transformations using a formal approach [77][7]. A constraint-based model type is a triple made of a set of class identifiers, a set of associations, and a set of constraints over classes and associations. A model type is extracted from a reusable transformation written in a particular MT language, and tested over a specific metamodel to check for conformance. If it is conforming, then it qualifies for reusing the transformation.

[6] http://github.com/MDEGroup/totem

[7] It must be noted that constraint-based model types were proposed before typing requirements models (2014 vs 2017), which take inspiration from that work.

Running example. The constraint-based model type for the running example is the following:

$$
\begin{aligned}
Classes &= \{C, A1, A2\} \\
Assocs &= \{(roots, C, A1), \\
&\quad\ (subs, A1, A2), \\
&\quad\ (container, A2, C)\} \\
Constr &= name(C) = Container \wedge \\
&\quad\ upper(roots) = * \wedge \\
&\quad\ lower(subs) = 0 \wedge \\
&\quad\ upper(container) = 1
\end{aligned}
$$

In the original formulation [77], class names are the class identifiers, which precludes expressing anonymous classes (needed for the running example). Hence, we assume a function name(_) which returns the expected name of a class. Functions lower and upper indicate the required minimum and maximum cardinality of associations. Just like the DRM in Fig. 19, the constraint-based model type does not add a restriction on the lower cardinality of roots or container, or the upper cardinality of subs, as the transformation allows any value.

Tool support. To the best of our knowledge, there is currently no tooling for constraint-based model types.

5 Discussion

In this section, we first compare the presented MT reuse approaches (Section 5.1). Then, we discuss on the extent to which they cover the needs stated by the participants of the survey (Section 5.2). Finally, we provide guidelines formulated as a decision tree for selecting a transformation reuse approach (Section 5.3).

5.1 Comparing MT reuse approaches

Based on the classification shown in Table 1, in the following, we discuss the differences of analysed MT reuse approaches 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. The last two rows of the table correspond to features proposed by survey participants.

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, facet-oriented modeling and multilevel modeling sup-

Table 2: Comparison of model transformation reuse approaches

	Model-Typing	Concepts	A-posteriori	Facets	Multilevel	MT Patterns	MOps	Typing Reqs	Const. Types
Reusing existing MT (opportunistic)	slicing	slicing	free	free	promotion	N.A.	N.A.	type extraction	type extraction
Making a MT reusable (systematic)	abstract MM	generic MM (concept)	role MM	facet MM	deep MM	design patterns	operators	N.A.	N.A.
Customization technique	adapter + mappings	adapter + mappings	adapter + mappings	adapter + mappings	instantiation + invariants	mappings	adapter + mappings	auto	auto
Customization complexity	low to high	low to high	low to high	low to high	medium to high	low	medium	low	low
Customization expressiveness	high (polymorphic reuse)	medium-high (parametric reuse)	very high (multi-matching, dynamic)	very high (multi-matching, dynamic)	medium (instantiation and invariants)	low (limited matching)	medium-high (ports)	medium (anon. classes, open features)	medium-low (logic, constraints)
Execution cost overhead	evaluation of adapter at run-time	none (adapter injected in MT at compile-time)	evaluation of adapter at run-time	facet creation at run-time	traverse typing relations at run-time	none (MT code derived from pattern at compile-time)	evaluation of adapter at run-time	none	none
Property preservation guarantees	static typing, polymorphism	static typing, generics	dynamic typing, constraint solving	dynamic typing	static typing, multilevel	static typing, generative	static typing, generative	static typing	static typing
Participant suggestions									
MT parametrization with metamodels	any MM	any MM	any MM	any MM	any MM	any MM	any MM	implicit, extracted from MT	implicit, extracted from MT
Systematic modularization	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	mapping operators structure MT reuse	N.A.	N.A.

port 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 [63], 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 [39]) 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 and facet-oriented modeling, there is no specific technique to simplify opportunistic reuse, while for systematic reuse one can create a role metamodel [38] or a facet metamodel [41]. The primary goal of these metamodels is not object creation but retyping. MT patterns and MOps are only relevant for systematic reuse, where abstract patterns or reusable mapping operators are available to be applied on a specific metamodel. Finally, constraint-based model types and typing requirements models are mostly applicable to opportunistic reuse, since they extract typing requirements for metamodels from existing MTs.

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, facet-oriented modeling, MT patterns and MOps rely on syntactic mappings. When further customizations are required to apply rt in a particular context, model typing, concepts, a-posteriori typing, facet-oriented modeling and MOps also support the definition of explicit adapters. In the case of patterns, the developer must refine the generated MT code by hand if the mapping is not one-to-one. Multilevel modeling relies on instantiation to map the initial metamodel MM to the actual metamodel MM' one metalevel below, while OCL invariants can emulate attribute adapters. Constraint-based model types and typing requirements models do not require specifying explicit mappings from the typing requirements description to the specific metamodels. Instead, a satisfaction or refinement relation automatically assesses whether the specific metamodels qualify for MT reuse.

The complexity of the adapters depends on the syntactic distance between the initial and actual metamodels. Hence, the cost to specify an adapter can range from low to high accordingly. Multilevel modeling requires a special metamodeling architecture. MT patterns and MOps require an explicit definition of the mapping even in case of an isomorphic alignment. Instead, model typing, typing requirements models and constraint-based model types may infer the mapping automatically.

Regarding the expressiveness of the mapping customization, model typing relies on polymorphic reuse and concepts on parametric reuse. A-posteriori typing and facet-oriented modeling support in addition multi-matching (i.e., a model element can get several types) and dynamic typing. Multilevel modeling uses instantiation for customization. MT 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, a-posteriori typing and facet-oriented modeling evaluate the adapters when the MT is called, multilevel modeling follows a similar approach by traversing the typing relationships at run-time, and MOps translate the input models into the format expected by the reused MT before its execution. However, the added flexibility of dynamic model typing, instance-level specifications of a-posteriori typing, and facet-oriented modeling may incur run-time penalties, as object types are dynamically calculated by queries. The concept-based 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 [15], and then compiled into a specific MT language. Similarly, constraint-based model types and typing requirements models do not incur any execution overhead either, as they just provide checks assessing correct reuse for specific metamodels.

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 both MT patterns and MOps use a generative approach. A-posteriori typing and facet-oriented modeling may use constraint solving at design-time to discard potentially unsafe matchings, but the correctness guarantees are limited by the bounded search of the constraint solver [38]. Typing requirements models and constraint-based model types use typing rules to extract a specification of the typing needs, to be checked against specific metamodels.

Finally, parameterization with metamodels is supported by all MT reuse approaches, while systematic modularization is only supported by MOps.

Altogether, the analysed MT reuse 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 (e.g., in model typing) or dynamically (e.g., 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 elements in the 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" [49,77]) 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 mostly 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" [16] or "flexible instantiation policies" [53]. More generally, automated mechanisms for composing a MT out of reused partial MTs are not exploited by the analysed approaches. This is so as all approaches – except MT patterns and MOps when used to build reusable model-to-model transformations – see the reused MT as a black box. In patterns and MOps, 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 [29,60].

5.2 Comparing MT reuse approaches and community needs

In the following, we compare current reuse techniques mentioned by the survey participants (cf. Section 2.2). Moreover, we also analyse how the reuse approaches cover the features that researchers demand from an ideal MT reuse approach, as indicated in the survey.

Reuse techniques that participants reported they have used include copy-paste, creating adapter transformations, early modularization and higher-order transformations. As previously mentioned, copy-paste requires manual adaptation of the copied transformation, and hence it is time consuming and error prone. As this was the most used option, it motivates the need for more powerful reuse techniques. Creating an adapter transformation (cf. Figure 2) also requires high effort. Regarding the features in Table 2, the customization complexity would be medium to high, while the expressiveness is very high (as arbitrary adaptations can be expressed using a transformation). The execution cost can be high, as an additional transformation needs to be executed, while the adapter transformation needs to be verified to ensure property preservation guarantees. Early modularization helps in making transformations easier to understand and maintain, but is not per se a technique to reuse a transformation for a different metamodel. Finally, higher-order transformations are a means to rewrite transformations, used for example by the concepts and the design patterns reuse approaches.

Next, we analyse the degree in which the 9 reuse approaches compared in Section 4 cover the desired features by the survey participants. Table 3 summarizes the desired features ordered by popularity with respect to the survey. The first seven features were fixed in the questionnaire, while the last two rows correspond to features freely suggested by the participants using an open text field. Overall, no existing approach satisfies all features, but they only provide between two and five features.

The most desired feature is being able to reuse parts of a transformation and not necessarily the whole transformation. While some MT languages provide some notion of modularization [17,72], we are not aware of any approach that slices parts of an existing model transformation not modularized from the beginning, to allow the reuse of those parts.

All approaches but MT patterns and MOps provide some support ensuring syntactical correct reuse. However, as mentioned in Section 5.1, no approach supports guarantees for semantically correct reuse, e.g., based on transformation intents.

All approaches offer a declarative way to reuse. This is usually based on establishing mappings, which in the case of model typing can be inferred. Typing requirements models

Table 3: Coverage of practitioner desired features by MT reuse approaches.

	Model-Typing	Concepts	A-posteriori	Facets	Multilevel	MT Patterns	MOps	Typing Reqs	Const. Types
Reusing parts of a transformation [22%]	✗	✗	✗	✗	✗	✗	✗	✗	✗
Analysis mechanisms to ensure correct reuse [17%]	✓	✓	✓	✓	✓	✗	✗	✓	✓
Declarative way to reuse [16%]	✓	✓	✓	✓	✓	✓	✓	✓	✓
Independence of the MT language [14%]	✓	✗	✓	✓	∼	✓	✓	✗	✗
Discovering reusable MTs for a metamodel [14%]	✓	✗	✗	✗	✗	✗	✗	✓	✓
Black box approach [13%]	✓	✓	✓	✓	✓	✗	✗	✓	✓
Low execution cost overhead [5%]	∼	✓	∼	∼	∼	✓	∼	✓	✓
Participant suggestions									
MT parametrization with metamodels [2%]	✓	✓	✓	✓	✓	✓	✓	∼	∼
Systematic modularization [2%]	✗	✗	✗	✗	✗	✗	∼	✗	✗

and constraint-based model types do not even require specifying any mapping.

Model typing, a-posteriori typing, facet-oriented modeling, MT patterns and MOps are independent from the MT language. In the first three cases, this is possible because the mechanisms (polymorphism, retyping) act on the metamodeling framework itself. Instead, MT patterns and MOps implement a generative approach, hence they can target new MT languages by defining suitable code generators for them. Concepts, typing requirements models and constraint-based model types are dependent on the MT language since they need to manipulate or analyse the MT code to be reused. Multilevel modeling is an in-between case because the mechanism acts on the metamodeling framework, but the MT language needs to be multilevel aware for better usability, as shown in [3,44].

Only model typing, typing requirements models and constraint-based model types provide support for discovering reusable MTs that can be applied on a given metamodel. For this purpose, they rely on automatic mechanisms to infer subtyping or conformance relations. In contrast, all other approaches require the user to explicitly specify mappings. In the case of typing requirements models and constraint-based model types, reuse discovery is supported by extracting the requirements of the reusable MT, and then checking whether the given metamodel satisfies them.

Most approaches are black-box, except MT patterns and MOps which require knowing the content of the MT to be reused. While black-box reuse is desirable in many cases, white-box reuse may provide a better understanding of the behavior of the reused MT, hence improving the chances of a semantically correct reuse.

As discussed in Section 5.1, concepts, MT patterns, typing requirements models and constraint-based model types have no execution overhead, while in the remaining cases, the overhead is moderate. Interestingly, very few participants in the survey considered that this feature was important in a MT reuse approach.

While the previous seven features were fixed in the questionnaire, participants could suggest other features that they considered important. Some of their suggestions are relevant to the *process* of reusing MTs, such as support for debugging, evolution upon metamodel changes, integration with the modeling editor, and testing transformations after reuse. In this paper, we are more concerned about MT reuse *techniques*, for which the features in the last two rows of Table 3 were mentioned.

The first suggested feature is the ability to parametrize the MT for arbitrary metamodels. All approaches support this in some way or another, as they are adaptable to metamodels different from the ones they were initially defined on. Such a "parameter" metamodel is defined explicitly in all approaches except in typing requirements models and constraint-based model types, where it is extracted from the transformation (see Table 2).

The second suggestion is to support systematic modularization of MTs, so that they can be sliced and then called externally. This is related to the reuse of parts of a transformation, and is hardly supported by any of the approaches. The only exception is MOps, which can be used to structure a reusable transformation in terms of mapping operators (see Table 2). However, in the running example, MOps are used differently to adapt a specific metamodel to the transformation metamodel, after which the complete transformation and not just a part is executed.

Overall, among all features demanded by the survey participants, the ones with less coverage by current MT reuse approaches are: reusing sliced parts of an existing transformation, discovering reusable transformations, and systematic modularization techniques. These functionalities,

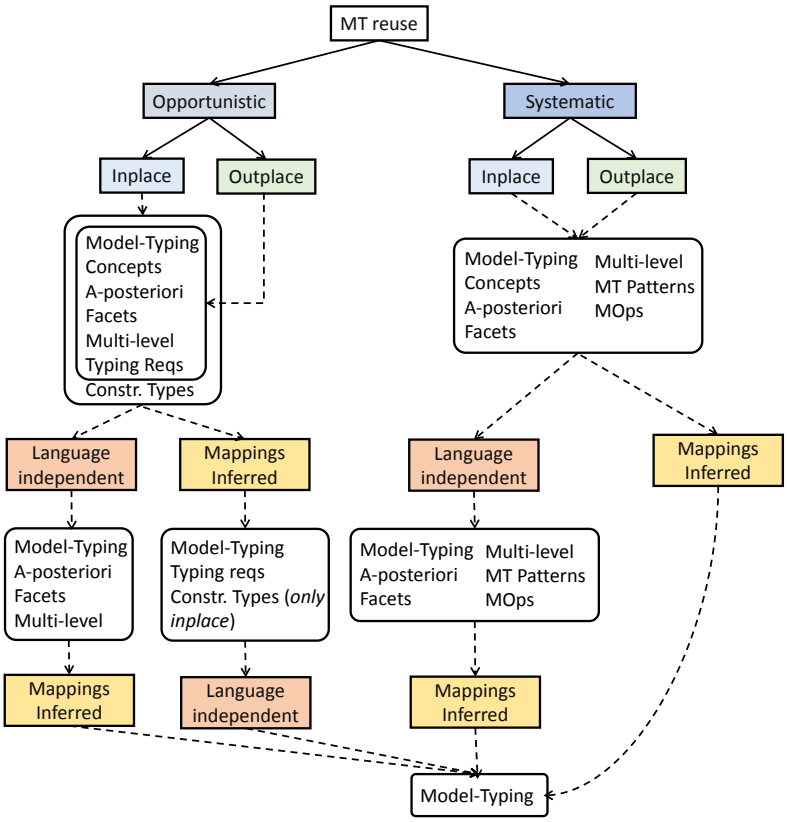


Fig. 20: A decision tree for choosing a transformation reuse approach

together with the uncharted features in the feature model (user-defined evaluation points for mapping execution (subfeature of feature 2b), intra-level mappings (subfeature of feature 2c), reuse based on intents (subfeature of feature 5a), and multiple reuse occurrences (feature 2e)) can be put forward to the community as promising research areas in MT reuse.

Finally, some reuse approaches are language-independent, and so they work with any MT language. For those techniques specific to a MT language (e.g., ATL), the main requirement is the availability of facilities to query, traverse and rewrite transformations. A representation of the transformation as a model conformant to a metamodel greatly helps in these tasks [62]. Therefore, this is a suggestion for future or existing MT languages to provide better support for reusability.

5.3 Guidelines for choosing a MT reuse approach

Next, we provide some guidelines to help developers choose a specific MT reuse approach depending on their needs. The guidelines are formulated as the (partial) decision tree shown in Fig. 20. This considers the most relevant aspects of MT reuse approaches, extracted from Table 1. The guidelines provide an efficient way to navigate through Table 1, where no specific order to the features is given. In the tree, colored boxes represent decisions, and rounded rectangles contain suitable reuse approaches for the selected decision. Boxes for decisions on the same concern (e.g., Language independent) have the same color.

As a first step, the developer needs to decide the intended kind of reuse: opportunistic or systematic. The first case occurs when there is a need to reuse an existing transformation for a new context. The second case typically arises when a MT being developed is expected to be reused many times in the future, e.g., because it will be available in a public repository or internally to a company, or because it implements recurrent behavior.

In a second step, the developer would identify whether the reused transformation is inplace or outplace. As Fig. 20 shows, this only has an impact for opportunistic reuse, because constraint-based model types do not currently support outplace transformations; the rest of approaches support both transformation kinds.

Next, the developer can look at two other orthogonal concerns in any order: whether independence of the transformation language is required, and whether mappings must be automatically inferred so that the developer does not need to explicitly set bindings, likely at the cost of lower expressiveness (i.e., there may be less opportunities for reuse). While

choosing either concern leads to a reasonable choice of approaches, both concerns are only supported by the model typing approach.

Independence of the transformation language may be desirable in systematic reuse when the reusable MT is built from scratch, as it increases the reuse opportunities by supporting various specific transformation languages. However, in opportunistic reuse, the choice of the transformation language has already been made; therefore, the question in this case would rather be whether the reuse approach supports (or is required to support) other MT languages than ATL, as the approaches left-out, concepts and typing requirements models are specific to ATL.

6 Related work

Reuse of MDE-related artefacts, like metamodels [36] and DSLs [12,44,68], is being actively investigated within the MDE community. In this paper, we have focused on reuse of transformations across metamodels, so-called inter-transformations in [32]. 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 [32] for further details on these kinds of reuse.

Intra-transformation reuse is typically specific for a MT language. Some of the proposed techniques include rules with variability [67], ATL module superimposition [17, 72], and rule inheritance [75]. Other internal composition mechanisms are phases [61], hooks [60] and unit combinators [29]. As discussed in Section 5, an interesting line of research is the combination of inter- and intra-transformation reuse.

Several classifications of MT approaches [11] and tools [26] exist. The features of some MT approaches, like *parameterization* or support for higher-order transformations, facilitate reuse. Most reuse approaches are independent of the MT language. However, those that are dependent (like concepts [62]) benefit from the declarative style of the MT language, as it simplifies the rewriting of the MT specification.

Similar to model typing [21,65], Boronat [8] proposes a method to discover subtyping (subsumption) relations between two metamodels. This method takes into consideration OCL constraints, and supports structural subtyping.

Several approaches build on the idea of genericity and reflection to make transformations configurable. In particular, generic MTs [71] are similar to concepts, but specifying relations between the type parameters is not possible, and there is limited support for adaptation [71]. In [47], the authors propose generic rules as a compact mechanism to specify modeling guidelines for Matlab/Simulink models. Generic rules can receive feature and class names as attributes, and make use of reflection to check the type of model elements.

Design patterns for MTs are gaining popularity since the past decade, yet their awareness is still low, as reported in [35]. For example, a catalog of MT design patterns is presented in [33,34], together with illustrative implementations of the patterns in UML-RSDS. This catalog is revised in [15], and an approach called DelTa to instantiate the patterns using higher-order transformations is proposed.

A widely used technique for software reuse is based on defining and connecting pre-built software units or components [46]. Component-based software development has been applied to several domains [54,58], but only incipiently to MDE and MTs [7,62]. In [62], the authors define transformation components that have concepts as their interface. Such components can be connected to each other via bindings. Regarding transformation composition, many languages have been proposed to specify transformation chains, like Wires [57], UniTI [70], MCC [29] or MTC Flow [2]. However, in most cases, the metamodels the transformations are defined on are fixed and cannot be reused for other metamodels. An exception is [6] where, given a transformation chain, an adapter transformation is produced in cases where the intermediate metamodels are incompatible but have resolvable heterogeneities.

All reuse approaches revised in this paper can be considered *open*, in the sense that they permit reusing a MT with a metamodel provided by the developer. Instead, other approaches support *closed* reuse, that is, they allow reusing a transformation for a closed metamodel set [40,56]. The idea of these approaches is to build a product line of metamodels [20], over which a transformation product line is defined. This way, transformations so defined become applicable for all metamodels in the set. Open reuse is more flexible but it typically requires specifying a binding between the transformation interface and the metamodel. Instead, closed reuse requires less effort which normally amounts to providing a configuration, but the set of metamodels a transformation can be reused on is fixed a priori.

7 Conclusion and perspectives

To achieve true engineering of MDE solutions, mechanisms to scale them up to industrial practice are required. This includes the development of reuse techniques for modeling artefacts – including MTs – as well as mechanisms that facilitate the maintenance and (co-)evolution of metamodels, models, MTs, code generators, etc.

In this paper, we have polled MT users through a survey to motivate our comparison, and analysed 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 [59], and the combination of intra- and inter-transformation reuse approaches. MT reuse is a real need, but existing reuse mechanisms have not percolated practice yet. While existing reuse approaches accomplish most desiderata of the community, some challenges remain, like partial transformation reuse, modularization techniques, and methods to discover transformations applicable to a given metamodel. Finally, we provide a decision tree to help developers select the reuse approach that fits their needs.

In the future, we plan to open the spectrum to other reuse scenarios. Expanding our survey with more industry participants, analysing how often MTs are reused in practice, and detecting reuse opportunities, e.g., using tools like [52], remain as future work.

Acknowledgements. We are grateful to all participants in the survey for their support, and to all participants in the 1st workshop on Unifying Software Reuse at Bellairs for their feedback on the feature model. This work has been partially funded by the Spanish Ministry of Science (RTI2018-095255-B-I00) and the Madrid Region (S2018/TCS-4314).

References

1. Survey raw material. `http://bit.ly/bellairs18`. Accessed: 2018-11-11
2. Alvarez, C., Casallas, R.: MTC Flow: A tool to design, develop and deploy model transformation chains. In: ACadeMics Tooling with Eclipse Workshop. ACM (2013). See also `http://www.mtcflow.com/`
3. Atkinson, C., Gerbig, R., Tunjic, C.V.: Enhancing classic transformation languages to support multi-level modeling. Software and Systems Modeling **14**(2), 645–666 (2015)
4. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Transactions on Modeling and Computer Simulation **12**(4), 290–321 (2002)
5. Babur, Ö., Cleophas, L., van den Brand, M., Tekinerdogan, B., Aksit, M.: Models, more models, and then a lot more. In: Software Technologies: Applications and Foundations, *LNCS*, vol. 10748, pp. 129–135. Springer (2017)
6. Basciani, F., Ruscio, D.D., Iovino, L., Pierantonio, A.: Automated chaining of model transformations with incompatible metamodels. In: Model Driven Engineering Languages and Systems, *LNCS*, vol. 8767, pp. 602–618. Springer (2014)
7. Bendraou, R., Desfray, P., Gervais, M.P., Muller, A.: MDA tool components: a proposal for packaging know-how in Model Driven Development. Software and Systems Modeling **7**(3), 329–343 (2008)
8. Boronat, A.: Structural model subtyping with OCL constraints. In: Software Language Engineering, pp. 194–205. ACM (2017)
9. Bruel, J.M., Combemale, B., Guerra, E., Jézéquel, J., Kienzle, J., de Lara, J., Mussbacher, G., Syriani, E., Vangheluwe, H.: Model transformation reuse across metamodels - A classification and comparison of approaches. In: Theory and Practice of Model Transformations, *LNCS*, vol. 10888, pp. 92–109. Springer (2018)
10. Criado, J., Martínez, S., Iribarne, L., Cabot, J.: Enabling the reuse of stored model transformations through annotations. In: Theory and Practice of Model Transformations, *LNCS*, vol. 9152, p. 15. Springer (2015)
11. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3), 621–645 (2006)
12. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Melange: A meta-language for modular and reusable development of DSLs. In: Software Language Engineering, pp. 25–36. ACM (2015)
13. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Safe model polymorphism for flexible modeling. Computer Languages, Systems and Structures **49**, 30 (2016)
14. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and kleisli categories. In: Fundamental Approaches to Software Engineering, *LNCS*, vol. 7212, pp. 163–177. Springer (2012)
15. Ergin, H., Syriani, E., Gray, J.: Design pattern oriented development of model transformations. Computer Languages, Systems & Structures **46**, 106–139 (2016)
16. Etien, A., Muller, A., Legrand, T., Paige, R.F.: Localized model transformations for building large-scale transformations. Software and Systems Modeling **14**(3), 1189–1213 (2015)
17. Fleck, M., Troya, J., Kessentini, M., Wimmer, M., Alkhazi, B.: Model transformation modularization as a many-objective optimization problem. IEEE Transactions on Software Engineering **43**(11), 1009–1032 (2017)
18. Gregor, D.P., Järvi, J., Siek, J.G., Stroustrup, B., Reis, G.D., Lumsdaine, A.: Concepts: linguistic support for generic programming in C++. In: Object-Oriented Programming, Systems, Languages, and Applications, pp. 291–310. ACM (2006)
19. Group, O.M.: Query/View/Transformation Specification, 1.3 edn. (2016). `https://www.omg.org/spec/QVT/About-QVT/`
20. Guerra, E., de Lara, J., Chechik, M., Salay, R.: Analysing metamodel product lines. In: Software Language Engineering, pp. 159–172. ACM (2018)
21. Guy, C., Combemale, B., Derrien, S., Steel, J., Jézéquel, J.M.: On model subtyping. In: European conference on Modelling Foundations and Applications, *LNCS*, vol. 7349, pp. 400–415. Springer (2012)
22. Jézéquel, J., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of metalanguages and its implementation in the kermeta language workbench. Software and System Modeling **14**(2), 905–920 (2015)
23. Jézéquel, J.M., Barais, O., Fleurey, F.: Model driven language engineering with Kermeta. In: Generative and Transformational Techniques in Software Engineering III, *LNCS*, vol. 6491, pp. 201–221. Springer (2011)
24. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1-2), 31–39 (2008)
25. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: International Conference on Software Engineering, pp. 485–495. IEEE Computer Society (2009)
26. Kahani, N., Bagherzadeh, M., R. Cordy, J., Dingel, J., Varro, D.: Survey and classification of model transformation tools. Software and Systems Modeling pp. 1–37 (2018)
27. Kalliamvakou, E., Palyart, M., Murphy, G.C., Damian, D.E.: A field study of modellers at work. In: International Workshop on Modeling in Software Engineering, pp. 25–29. IEEE Computer Society (2015)
28. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
29. Kleppe, A.: MCC: A model transformation environment. In: European Conference on Modelling Foundations and Applications, *LNCS*, vol. 4066, pp. 173–187. Springer (2006)

30. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Theory and Practice of Model Transformations, *LNCS*, vol. 5063, pp. 46–60. Springer (2008)
31. Krueger, C.W.: Software reuse. ACM Computer Surveys **24**(2), 131–183 (1992)
32. Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: Reuse in model-to-model transformation languages: are we there yet? Software and Systems Modeling **14**(2), 537–572 (2015)
33. Lano, K., Rahimi, S.K.: Model-transformation design patterns. IEEE Transactions on Software Engineering **40**(12), 1224–1259 (2014)
34. Lano, K., Rahimi, S.K., Poernomo, I., Terrell, J., Zschaler, S.: Correct-by-construction synthesis of model transformations using transformation patterns. Software and Systems Modeling **13**(2), 873–907 (2014)
35. Lano, K., Rahimi, S.K., Tehrani, S.Y., Sharbaf, M.: A survey of model transformation design patterns in practice. Journal of Systems and Software **140**, 48–73 (2018)
36. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. Software and Systems Modeling **12**(3), 453–474 (2013)
37. de Lara, J., Guerra, E.: Towards the flexible reuse of model transformations: A formal approach based on graph transformation. Journal of Logical and Algebraic Methods in Programming **83**(5-6), 427–458 (2014)
38. de Lara, J., Guerra, E.: A posteriori typing for model-driven engineering: Concepts, analysis, and applications. ACM Transactions on Software Engineering and Methodology **25**(4), 31:1–31:60 (2017)
39. de Lara, J., Guerra, E.: Refactoring multi-level models. ACM Transactions on Software Engineering and Methodology **27**(4), 17:1–17:56 (2018)
40. de Lara, J., Guerra, E., Chechik, M., Salay, R.: Model transformation product lines. In: Model Driven Engineering Languages and Systems, pp. 67–77. ACM (2018)
41. de Lara, J., Guerra, E., Kienzle, J., Hattab, Y.: Facet-oriented modelling: Open objects for model-driven engineering. In: Software Language Engineering, pp. 146–158. ACM (2018)
42. de Lara, J., Guerra, E., di Ruscio, D., di Rocco, J., Sánchez Cuadrado, J., Iovino, L., Pierantonio, A.: Automated reuse of model transformations through typing requirements models. ACM Transactions on Software Engineering and Methodology **28**(4), 21:1–21:62 (2019)
43. de Lara, J., Guerra, E., Sánchez Cuadrado, J.: When and how to use multilevel modelling. ACM Transactions on Software Engineering and Methodology **24**(2), 12:1–12:46 (2014)
44. de Lara, J., Guerra, E., Sánchez Cuadrado, J.: Model-driven engineering with domain-specific meta-modelling languages. Software and Systems Modeling **14**(1), 429–459 (2015)
45. de Lara, J., di Rocco, J., di Ruscio, D., Guerra, E., Iovino, L., Pierantonio, A., Sánchez Cuadrado, J.: Reusing model transformations through typing requirements models. In: Fundamental Approaches to Software Engineering, *LNCS*, vol. 10202, pp. 264–282. Springer (2017)
46. Lau, K., Wang, Z.: Software component models. IEEE Transactions on Software Engineering **33**(10), 709–724 (2007)
47. Legros, E., Amelunxen, C., Klar, F., Schürr, A.: Generic and reflective graph transformations for checking and enforcement of modeling guidelines. Journal of Visual Languages & Computing **20**(4), 252–268 (2009)
48. Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J.: Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. Software and System Modeling **17**(1), 91–113 (2018)
49. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G.M., Syriani, E., Wimmer, M.: Model transformation intents and their properties. Software and Systems Modeling **15**(3), 685–705 (2014)
50. Macías, F., Rutle, A., Stolz, V., Rodríguez-Echeverría, R., Wolter, U.: An approach to flexible multilevel modelling. Enterprise Modelling and Information Systems Architectures **13**, 10:1–10:35 (2018)
51. Mengerink, J.: The DSL/model co-evolution problem in industrial MDE ecosystems. Ph.D. thesis, Department of Mathematics and Computer Science (2018). Proefschrift
52. 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: International Workshop on Software Measurement, IWSM-Mensura, pp. 116–121. ACM (2017)
53. Morin, B., Klein, J., Kienzle, J., Jézéquel, J.M.: Flexible model element introduction policies for aspect-oriented modeling. In: Model Driven Engineering Languages and Systems, *LNCS*, vol. 6395, pp. 63–77. Springer (2010)
54. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. Computer **33**(3), 78–85 (2000)
55. Paige, R.F., Kolovos, D.S., Rose, L.M., Drivalos, N., Polack, F.A.C.: The design of a conceptual framework and technical infrastructure for model management language engineering. In: International Conference on Engineering of Complex Computer Systems, pp. 162–171. IEEE Computer Society (2009)
56. Perrouin, G., Amrani, M., Acher, M., Combemale, B., Legay, A., Schobbens, P.Y.: Featured model types: towards systematic reuse in modelling language engineering. In: Workshop on Modeling in Software Engineering, pp. 1–7. IEEE (2016)
57. Rivera, J.E., Ruiz-Gonzalez, D., Lopez-Romero, F., Bautista, J., Vallecillo, A.: Orchestrating ATL model transformations. In: MtATL 2009, pp. 34–46 (2009)
58. Saks, K.: JSR 318: Enterprise java beans, version 3.1. `http://download.oracle.com/otndocs/jcp/ejb-3.1-mrel-evalu-oth-JSpec/` (2009)
59. Salay, R., Zschaler, S., Chechik, M.: Correct reuse of transformations is hard to guarantee. In: Theory and Practice of Model Transformations, *LNCS*, vol. 9765, pp. 107–122. Springer (2016)
60. Sánchez Cuadrado, J., García Molina, J.: Approaches for model transformation reuse: Factorization and composition. In: Theory and Practice of Model Transformations, *LNCS*, vol. 5063, pp. 168–182. Springer (2008)
61. Sánchez Cuadrado, J., García Molina, J.: Modularization of model transformations through a phasing mechanism. Software and Systems Modeling **8**(3), 325–345 (2009)
62. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: A component model for model transformations. IEEE Transactions on Software Engineering **40**(11), 1042–1060 (2014)
63. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Reverse engineering of model transformations for reusability. In: Theory and Practice of Model Transformations, *LNCS*, vol. 8568, pp. 186–201. Springer (2014)
64. Sendall, S., Kozaczynski, W.: Model transformation: the heart and soul of model-driven software development. IEEE Software **20**(5), 42–45 (2003)
65. Steel, J., Jézéquel, J.M.: On model typing. Software and Systems Modeling **6**(4), 401–414 (2007)
66. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data Knowl. Eng. **35**(1), 83–106 (2000)
67. Strüber, D., Rubin, J., Arendt, T., Chechik, M., Taentzer, G., Plöger, J.: Variability-based model transformation: formal foundation and application. Fundamental Approaches to Software Engineering **30**(1), 133–162 (2018)
68. Sutîi, A.M., 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**, 48–70 (2018)

69. Van Gorp, P.: Applying traceability and cloning techniques to compose input-destructive model transformations into an input-preserving chain. In: Workshop on Composition and Evolution of Model Transformations. King's College (2011)
70. Vanhooff, B., Ayed, D., Baelen, S.V., Joosen, W., Berbers, Y.: UniTI: A unified transformation infrastructure. In: Model Driven Engineering Languages and Systems, *LNCS*, vol. 4735, pp. 31–45. Springer (2007)
71. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: The Unified Modeling Language. Modeling Languages and Applications, *LNCS*, vol. 3273, pp. 290–304. Springer (2004)
72. Wagelaar, D., Straeten, R.V.D., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Software and Systems Modeling **9**(3), 285–309 (2010)
73. Whittle, J., Hutchinson, J.E., Rouncefield, M.: The state of practice in model-driven engineering. IEEE Software **31**(3), 79–85 (2014)
74. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Surviving the heterogeneity jungle with composite mapping operators. In: Theory and Practice of Model Transformations, *LNCS*, vol. 6142, pp. 260–275. Springer (2010)
75. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Kolovos, D.S., Paige, R.F., Lauder, M., Schürr, A., Wagelaar, D.: Surveying rule inheritance in model-to-model transformation languages. Journal of Object Technology **11**(2), 3:1–46 (2012)
76. Wuliang, S., Combemale, B., Derrien, S., France, R.: Using model types to support contract-aware model substitutability. In: European Conference on Modelling Foundations and Applications, *LNCS*, vol. 7949, pp. 118–133. Springer (2013)
77. Zschaler, S.: Towards constraint-based model types: A generalised formal foundation for model genericity. In: Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, pp. 11–18. ACM (2014)