Universidad Autónoma de Madrid

Biblos-e Archivo
Repositorio Institucional UAM

Repositorio Institucional de la Universidad Autónoma de Madrid
https://repositorio.uam.es

Esta es la **versión de autor** del artículo publicado en:
This is an **author produced version** of a paper published in:

Juan De Lara, Esther Guerra, and Jörg Kienzle. 2021. Facet-oriented Modelling. ACM Trans. Softw. Eng. Methodol. 30, 3, Article 27 (February 2021), 59 pages.

DOI: https://doi.org/ 10.1145/3428076

Copyright: © 2021 Association for Computing Machinery

El acceso a la versión del editor puede requerir la suscripción del recurso
Access to the published version may require subscription

Facet-oriented modelling

JUAN DE LARA, Universidad Autónoma de Madrid (Spain)
ESTHER GUERRA, Universidad Autónoma de Madrid (Spain)
JÖRG KIENZLE, McGill University (Canada)

Models are the central assets in model-driven engineering (MDE), as they are actively used in all phases of software development. Models are built using metamodel-based languages, and so, objects in models are typed by a metamodel class. This typing is static, established at creation time, and cannot be changed later. Therefore, objects in MDE are *closed* and fixed with respect to the class they conform to, the fields they have, and the wellformedness constraints they must comply with. This hampers many MDE activities, like the reuse of model-related artefacts such as transformations, the opportunistic or dynamic combination of metamodels, or the dynamic reconfiguration of models.

To alleviate this rigidity, we propose making model objects *open* so that they can acquire or drop so-called *facets*. These contribute with a type, fields and constraints to the objects holding them. Facets are defined by regular metamodels, hence being a lightweight extension of standard metamodelling. Facet metamodels may declare usage *interfaces*, as well as *laws* that govern the assignment of facets to objects (or classes).

This paper describes our proposal, reporting on a theory, analysis techniques and an implementation. The benefits of the approach are validated on the basis of five case studies dealing with annotation models, transformation reuse, multi-view modelling, multi-level modelling and language product lines.

Categories and Subject Descriptors: [**Software and its engineering**]: Model-driven software engineering; Domain specific languages; Design languages; Software design engineering

Additional Key Words and Phrases: Metamodelling, Flexible Modelling, Role-Based Modelling, METADEPTH

ACM Reference Format:
Juan de Lara, Esther Guerra, Jörg Kienzle. 2019. Facet-oriented modelling. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 58 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Model-driven engineering (MDE) is a paradigm for software construction that fosters an active use of models throughout the software development process [Schmidt 2006]. This way, models in MDE serve as abstractions and automation devices for many development tasks, being used to specify, simulate, verify, generate code and maintain the final system, among other activities. Models are created using a modelling language, either general-purpose – like UML [UML 2017] – or domain-specific [Kelly and Tolvanen 2008]. In MDE, modelling languages are defined by a metamodel, a model itself that contains the types of elements (classes, associations, attributes) that can be used to create the models, and the constraints that the models should obey.

In standard MDE, classes in metamodels are used as templates to create objects. This means that they dictate the structural properties that the objects of a class have and

Author's addresses: J. de Lara, Computer Science Department, Universidad Autónoma de Madrid (Spain); E. Guerra, Computer Science Department, Universidad Autónoma de Madrid (Spain); J. Kienzle, Computer Science Department, McGill University (Canada).
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1049-331X/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

the wellformedness constraints they must comply with. When an object is created, it acquires its type, fields and constraints from its class, and these cannot change afterwards. This complicates some MDE activities. One of them is reuse since, for example, a model transformation t defined over a metamodel MM cannot be directly applied to instances of other metamodels [Bruel et al. 2020]. However, if we could assign types from MM to the instances of other metamodels, we could reuse the transformation t as is on those instances. Another case is model extension, i.e., increasing objects with additional fields. This is useful to insert auxiliary model elements needed by a transformation [Kolovos et al. 2010], or to enable layered modelling (i.e., building models incrementally according to predefined information layers) [Hendrickson et al. 2006]. These scenarios typically require merging the metamodels that define the allowed extensions (e.g., the auxiliary or layer elements) a priori, which precludes the opportunistic and dynamic addition of new information.

There are several proposals to enable more flexible modelling, most notably *a-posteriori typing* and *role-based modelling*. In a-posteriori typing [de Lara and Guerra 2017], objects can be assigned new types, but cannot acquire or drop new features dynamically. In role-based modelling [Bachman and Daya 1977; Steimann 2000; Kühn et al. 2015], objects can acquire and drop *roles* dynamically, and such roles are typed by role types and may have attributes. Still, role-based approaches lack declarative means to specify how the roles are to be acquired or dropped by objects, how the fields provided by a role relate to the object fields, or to impose additional constraints. Moreover, roles must be designed a priori, using new constructs that increase accidental complexity and are often difficult to integrate within existing metamodelling frameworks.

To overcome these problems we propose a new lightweight mechanism to bring dynamism and flexibility to modelling: the *facets*. The approach is inspired by the notion of roles and capitalizes on existing MDE works on metamodel extension and model adaptation [Brunelière et al. 2015a; Madiot and Dupe 2018; Langer et al. 2012; Barbero et al. 2007; Kolovos et al. 2006b; Brunelière et al. 2015b]. Our proposal goes further than these approaches by considering objects as *open* elements. This way, objects can dynamically acquire or drop types, features and constraints, which are provided by facets. Facets are regular objects, so there is no need to introduce new concepts in practice, and this is why we term our approach *lightweight*. We propose a new kind of specification – facet laws – that govern when objects can take other objects as facets, favouring separation of concerns [Dijkstra 1982]. Moreover, the value of fields of objects and facets can be synchronized, and facets can be applied at any metalevel.

The approach is backed by a theory, while analysis techniques based on model finding permit discovering conflicts between the conditions to acquire and drop facets, and the metamodel integrity constraints. The approach is implemented on top of our metamodelling tool METADEPTH [de Lara and Guerra 2010]. This tool is integrated with the Epsilon language family [Paige et al. 2009], which includes languages for in-place and model-to-model transformation and for code generation. Our implementation permits their use taking into account the semantics of facets. As a validation, we show how facets can tackle several MDE scenarios that normally would require building dedicated tooling, like the handling of annotation models, the reuse of model-based services and transformations [Bruel et al. 2020], multi-view modelling [Bruneliere et al. 2019], multi-level modelling [de Lara et al. 2014] and language product lines [White et al. 2009].

Overall, the contributions of this paper are the following: (i) a new approach enabling objects to dynamically acquire or drop types, fields and constraints depending on specified declarative conditions, which generalizes standard modelling; (ii) a synchronization mechanism for acquired and owned fields which automates their update upon changes; (iii) an implementation of these concepts atop the METADEPTH tool; (iv) an evaluation

of the applicability of facets to solve six MDE scenarios. The latter evaluation aims at answering the research question underlying this work: *Can a dynamic mechanism for object adaptation be used to solve MDE scenarios that are currently tackled using dedicated, ad-hoc tooling?* To answer this question, we compare our approach based on facets with 25 other tools and approaches, most of them specifically built for one specific scenario. This analysis shows that the mechanism of facets could have been used instead of developing dedicated tools, which is an important finding for the MDE research community and tool builders.

This paper is a further development of our preliminary work presented at the Software Language Engineering 2018 conference (SLE'18) [de Lara et al. 2018b], which has been extended as follows. First, we provide a formal foundation for facets, with proofs included in the appendix. This theory generalizes the intuition provided in [de Lara et al. 2018b] supporting facets over both classes and objects. We have incorporated intersection types, and support for reactive field adapters in facet laws. We have developed analysis techniques based on model finding [Jackson 2006; Kuhlmann and Gogolla 2012] and realized them within the METADEPTH tool. Finally, we describe new applications of facets to product lines of modelling languages and multi-level modelling.

The rest of the paper is organized as follows. Section 2 motivates our proposal using a running example. Section 3 provides an overview of our proposal. Section 4 introduces the notion of facet and a formal theory. Section 5 describes techniques for handling facets. Section 6 defines two mechanisms to control how facets can be used: interfaces and laws. Section 7 proposes analysis mechanisms for facet laws based on model finding. Section 8 reports on tool support. Section 9 evaluates the approach on five scenarios and discusses strengths and limitations. Section 10 compares with related research, and Section 11 finishes with the conclusions and prospects for future work. An appendix includes the proofs of the lemmas in Section 4.2.

2. MOTIVATION

This section introduces a running example to support the presentation of our approach. While here we focus on an information integration scenario, Section 9 shows the benefits of facets to solve five other MDE scenarios as well.

Assume a city hall needs a registry of the people living in the city, and for that purpose it is using the Census metamodel depicted in Figure 1(a). The metamodel allows creating Person objects with a fullName, an age, a female flag to indicate gender, a mandatory address and optionally a spouse.

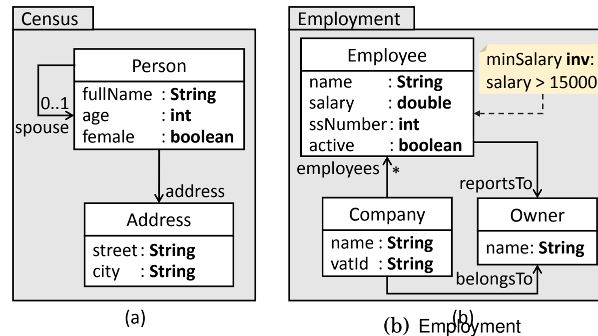


Fig. 1: Metamodels for the running example.

With the aim to provide tax calculation services to citizens, the city hall decides to increase the existing Census models with employment data. The structure of these data is captured by the metamodel in Figure 1(b). The intention is extending Person objects representing employed people with the fields in Employee or Owner, where Person.fullName corresponds to name in the other two classes. This extension should capture the dynamic nature of employments, so that objects representing unemployed people should lack information about employment.

w

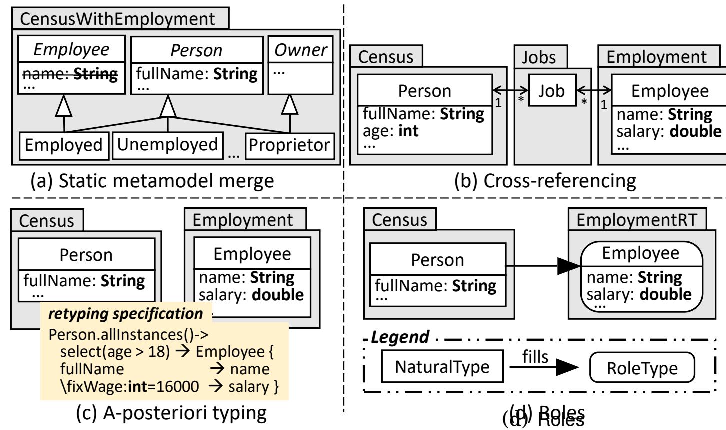


Fig. 2: Some alternative solutions for the running example.

Metamodel merging. One option is merging the metamodels in Figures 1(a) and 1(b) statically to obtain the metamodel in Figure 2(a) or a similar one, and then migrating any existing Census model to become an instance of this new metamodel. However, this solution has some drawbacks. First, it is not modular because it requires creating a new metamodel and resolving possible issues between attributes (e.g., Employee.name is deleted and superseded by Person.fullName). Second, model migration cannot be fully automated, because each Person object may have to be converted into Employed, Unemployed or company owner (Proprietor). Most importantly, this solution does not account for the dynamic nature of employments, where a same Person object should be able to take or leave employments without changing its identity, have several jobs, or be both owner and employee. While the latter can be allowed by adding a new subclass, the number of combinations may be exponential.

Views. We can tackle the scenario by means of views [Atkinson et al. 2015; Bruneliere et al. 2019]. For example, relying on a single underlying model (SUM), we could adopt a so-called *essential* approach [Meier et al. 2019]. This would require developing an integrated metamodel like the one in Figure 2(a), and then define projections onto two viewpoints defined by the Census and Employment metamodels of Figure 1. However, this solution has the same limitations as metamodel merging: lack of dynamicity for object types, fields and constraints.

Alternatively, using languages like ModelJoin [Burger et al. 2016], we can adopt a *pragmatic* approach whereby a view joins the information from the given Census and Employment models [Meier et al. 2019]. In the background, this solution creates

a merged metamodel on the fly, as well as transformations from the models to the view. However, the propagation of values from the view to the models needs to be programmed by hand, and the limitation of non-dynamicity of types remains.

Cross-referencing. Another option is to keep the metamodels in Figure 1 separate, and use a correspondence metamodel to declare cross-references between them, as Figure 2(b) shows. This enables dynamicity, as Employees can be created and deleted on demand, and be linked and unlinked to Person objects. However, this solution complicates management by adding a level of indirection, because accessing the salary of a Person or the age of an Employee requires navigating the correspondence links explicitly. For example, the salary of a Person object p would be obtained with an OCL [OCL 2014] expression like p.jobs→first().employee.salary. Instead, a solution supporting transparent access to the fields of Person and those acquired from Employee would use simpler expressions (e.g., p.salary in the given example)[1]. Finally, cross-referencing makes it necessary to write a program that maintains the models consistent, e.g., enforcing the fullName of a Person to be the same as the name of the Employee objects it refers to, whenever any of the two changes. Instead, a native solution for consistency is preferable.

A-posteriori typing. Figure 2(c) shows a solution based on a-posteriori typing. It requires writing a specification of the conditions for a Person object to be assigned Employee as additional type (in the figure, when age is bigger than 18). This additional typing is dynamic as it depends on the object properties, but it is not possible to assign the a-posteriori type Employee to a specific Person object manually, as required in this scenario. Moreover, each field in the a-posteriori type Employee needs to be backed on a field of Person or on a (read-only) derived attribute. This is why the specification creates the derived attribute fixWage mapped to salary, so that each Person earns the same and this cannot be changed because the attribute is derived. This is not satisfactory in our scenario, which requires adding new mutable fields to Person objects.

Roles. Figure 2(d) sketches a solution based on roles. It requires the ad-hoc construction of role types (e.g., Employee) that mimic the metamodel types in Figure 1(b). Some approaches [Kühn et al. 2015] require role types to reside in compartment types, but we omit this for simplicity. Then, it is necessary to declare that Person (a so-called *natural type*) may fill the role type Employee. Hence, this solution is heavyweight and intrusive as it relies on additional constructs. That is, the given metamodels cannot be used out of the box, but a new model that uses role types, natural types and other specialized constructs needs to be built. Finally, there is no way to express relations between role type attributes and natural type attributes (e.g., name should be equal to fullName), conditions for natural types to take role types (e.g., only adult Persons can be Employees), or facilities to incorporate role type constraints into natural types. Section 10 will present a more detailed analysis of roles.

Altogether, from the examined solutions to the presented scenario of information integration, we favour a solution that meets the following requirements:

R1. Be modular and non-intrusive, so that there is no need to create or change existing models or metamodels. This avoids having to migrate models, reorganize metamodels or modify them to introduce non-standard constructs (e.g., roles, fills-in

[1] We could avoid modifying the Census and Employee metamodels by having unidirectional references from Job to Person and Employee, instead of bidirectional associations. However, the navigation from Person objects to Employee objects would be more inefficient, as it would require iterating on all Job objects in the worst case. The navigation expressions would be more complicated as well: p.salary would become Job.allInstances()→any(j | j.person=p).employee.salary.

relations), so that the approach can be applied to existing modelling artefacts as they are.

R2. Allow objects to acquire new types, fields and constraints, likely specified in other metamodels, and make them transparently accessible (i.e., avoiding explicit navigation through correspondence links). This simplifies the code of the resulting solution, since acquired types and fields would become seamlessly accessible from constraints and transformation programs (i.e., we can write p.salary instead of p.jobs→first().employee.salary).

R3. Support both automatic and manual acquisition and loss of types, fields and constraints. The former is useful to allow a declarative specification of the conditions for an object to acquire and drop specific types. The latter enables a fine-grained control of the types assigned to an object.

R4. Support for specification and automatic maintenance of relations (e.g., equality, or user-defined relations) between owned and acquired fields. This avoids coding attribute update procedures to be triggered upon attribute changes.

While these requirements are generic (in the sense that they are about the capabilities of objects to acquire/drop fields and types dynamically, without changing existing metamodels), Section 9 will introduce more specific criteria for other scenarios that facets can handle. Table I summarizes the coverage of the requirements by the previous approaches. As none of the analysed solutions support all these requirements, the rest of the paper proposes facets as a solution to cover them.

Table I: Coverage of requirements by the analysed approaches.

Approach	**R1: modular, non-intrusive**	**R2: acquisition of new types, fields and constraints; their transparent access**	**R3: dynamicity, both manual and automatic**	**R4: automatic maintenance of relations between owned and acquired fields**
Metamodel merging	No: *requires creating new metamodel and migrating existing models*	Yes	No: *types are static*	No
Views/essential	No: *requires creating SUM metamodel and solving possible conflicts*	Yes	No: *types are static*	Yes
Views/pragmatic	Yes	Yes: *but views are read-only (by default)*	No: *types are static*	Yes
Cross-referencing	No: *requires creating correspondence metamodel*	No: *field access is not transparent*	Yes: *but requires encoding update programs*	No
A-posteriori typing	Yes	Yes: *but acquired fields are read-only*	Partially: *only automatic*	No: *acquired fields are read-only*
Roles	No: *requires creating role types*	Partially: *constraints cannot be acquired*	No	No

3. OVERVIEW OF FACET-ORIENTED MODELLING

Figure 3 depicts the high-level view of our facet-oriented modelling approach, introducing some terminology to facilitate the reading of the remainder of the paper. The figure includes the number of the section that explains each concept.

Facets are objects that extend other objects (called *host* objects) with a new type, fields and constraints. Facets are regular objects conformant to a regular standard metamodel, that we call *facet metamodel*. In turn, we term the metamodel the host objects are conformant to *creation metamodel*. Sections 4 introduces the basic working mechanism of facets, and Section 6.1 describes how to restrict the classes from a facet metamodel that can be used as facets, by means of *facet interfaces*.

Several management commands are available to users for adding and removing facets from host objects (cf. Section 5.1). The fields contributed by a facet can be assigned a value just at creation time, or else be synchronized with other field values by means of

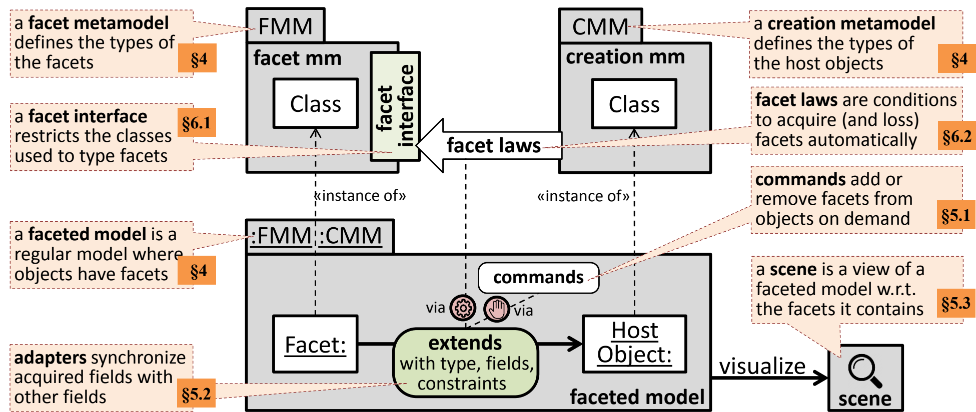


Fig. 3: Facet-oriented modelling at a glance.

reactive adapters (cf. Section 5.2). The defined adapters can be analysed to ensure the absence of infinite loops of field value computations (cf. Section 7.2). Models holding host objects with facets are called *faceted models*. These can be sliced according to various criteria to produce different types of *scenes*, which we introduce in Section 5.3.

In addition to commands, which must be issued by hand, it is possible to declare *facet laws* specifying conditions for the automated acquisition and loss of facets (cf. Section 6.2). Facet laws can be analysed for applicability, to detect situations in which the laws collide with the creation metamodel constraints preventing the creation of any faceted model (cf. Section 7.1).

4. FACETS

This section first introduces facets intuitively (Section 4.1) and then provides a formal theory (Section 4.2). The aim of the theory is to precisely define the working scheme of facets, to show that they generalize standard modelling, and to provide a blueprint for builders of new modelling frameworks incorporating the notion of facet.

4.1. Facets by example

We define facets as follows:

A facet is an object, which becomes part of another one(s), called the host object(s), such that the fields of the facet become transparently accessible from the host object, which also acquires the type and constraints of the facet. Facets are dynamic, and so a host object can acquire and drop facets dynamically.

Figure 4(a) shows an example of a host object homer with an attached facet of type Employee. A facet is an object that relates to other objects (their hosts) in a special way. This relation – represented with a dashed arrow – permits seeing an object and its facets as a whole, as depicted in Figure 4(b). The host object acquires the type and fields provided by the facet transparently. This means that, in Figure 4(b), homer.salary is a valid feature access, and the OCL query Employee.allInstances() yields Set{homer}. Despite Figure 4(b) shows the facet of the host object using physical containment, facets can be shared among objects, and an object can have several facets of the same or different type. If homer had additional facets, e.g., of type Owner, it would acquire their types and fields as well.

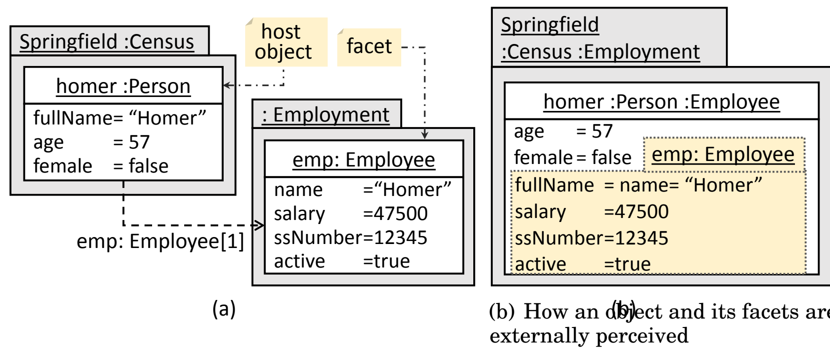


Fig. 4: Adding an Employee facet to the host object homer.

Facets also bring the constraints defined in their type to the host object. In the figure, the facet Employee adds constraint minSalary (cf. Figure 1(b)) to homer. Hence, the assignment homer.salary:=13000 makes object homer violate the constraint.

Facet fields are exposed by the host object, but can also be synchronized with the fields of the host. For example, in Figure 4(b), fullName (from Person) and name (from Employee) are required to be equal. Hence, the assignment homer.fullName:='Homer Simpson' causes homer.name to take the value 'Homer Simpson' as well. Likewise, changing home.name makes homer.fullName change its value in the same way. This equality relation is not restricted to be binary, but as facets can be shared, it may involve several objects. For example, assume that several people can share an employment because they work part-time. This can be modelled with various Person objects (say o1, o2 and o3) sharing the same facet of type Employee. This way, if we set o1.salary:=42567, the change will be observed on o1, o2 and o3 as it is made on the field salary of the shared facet. Section 5.2 will show how to set arbitrary reactive relations between fields, in addition to equality.

Our approach is agnostic on how host objects are created. In Figure 4(a), we use a creation metamodel (Census). While this is the standard approach to create objects in MDE, some systems support objects with no ontological type (i.e., do not require any class to create objects) [de Lara and Guerra 2010]. Such untyped objects may be host objects as well, acquiring their type from the facets. Figure 5 illustrates this situation, where homer is an untyped object with two facets of types Person and Employee. Under this view, the notion of object can be diminished to a *shell* providing an identity for the facets it
facets ca
of host o
facet met
used to c
both face

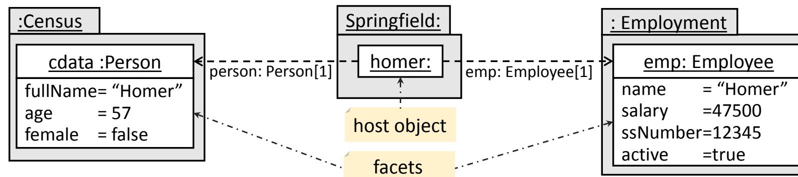


Fig. 5: Host object (homer) with no creation type but with two facets.

Similar to host objects, facets may or may not have an ontological type. Untyped facets are useful at the metamodel level (i.e., attached to classes) to share common attributes among a set of classes, similarly to inheritance. In Figures 4 and 5, facets are used at the model level to increase existing objects with additional fields, types and constraints.

As facet fields are reexposed as host object fields, there may be collisions of field names between a host object and its facets, or between different facets held by the same host object. To disambiguate, we provide a mechanism to refer to object facets explicitly. For example, assume object homer has two jobs, for which two Employee facets are created. Facets are named, so we can call the two facets dayJob and nightJob. Accessing homer.salary is ambiguous as it can refer to the field of either facet. Hence, the access should use the explicit facet name: homer.dayJob.salary or homer.nightJob.salary.

Finally, as facets are regular objects, we allow facets to have facets. However, we forbid cycles of facet relations.

Altogether, facets are modular and non-intrusive, satisfying requirement **R1** described in Section 2, while their characteristics can be seamlessly accessed from the host objects, satisfying **R2**.

4.2. Formal foundation of facets

Once we have provided an intuition of facets and their usage, this section presents their formal definition. Our goal is, on the one hand, to precisely define the meaning of facets, and on the other hand, to show that standard modelling can be seen as a special case of facet-oriented modelling.

In the following, given a relation r, we use r^+ to denote its transitive closure, r^* for its reflexive transitive closure, $r \rhd A$ for the restriction of its codomain (the range) to set A, and $A \lhd r$ for the restriction of its domain to set A.

We start by defining a faceted model.

Definition 4.1 (*Faceted model*). A faceted model is a tuple

$$M = \langle H, F, S, L, \mathit{feats}, \mathit{tar}, \mathit{facets} \rangle$$

made of:

— Disjoint sets H of host objects, F of facets, S of slots and L of links. We also define the sets $O = H \cup F$ of objects and $P = S \cup L$ of properties or features.
— A surjective relation $\mathit{feats} \subseteq F \times P$ assigning each facet a set of features.
— A function $\mathit{tar} \colon L \to F$ assigning each link a target facet.
— A surjective relation $\mathit{facets} \subseteq O \times F$ relating objects to facets, s.t. facets^+ is acyclic.

In this definition, only facets hold features (slots, links), while host objects aggregate facets. Facets can have facets, and every facet must be assigned at least to an object (and so facets is surjective). A facet can be shared among several objects, and so facets is a relation instead of a function. Feature ownership is captured by relation feats, which allows sharing features among facets. The definition abstracts from slot values as they are not relevant for the purpose of the theory.

Next, we define methods to access the features of a host object (feats_H) and the target host objects of a link (tar_H).

$$\mathit{feats}_H \subseteq H \times P \triangleq \mathit{feats} \circ \mathit{facets}^+ \tag{1}$$

$$\mathit{tar}_H \subseteq L \times H \triangleq (\mathit{facets}^{+^{-1}} \circ \mathit{tar}) \rhd H \tag{2}$$

Relation feats_H retrieves all features of the facets assigned directly and indirectly to a host object (hence the use of facets^+). It can be seen as a delegation-based mechanism

for feature access [Lieberman 1986]. Relation tar_H returns the host objects that directly or indirectly hold the facet a link points to. For this purpose, tar_H needs to recursively traverse $facets$ backwards (hence the $^+$ and $^{-1}$).

Example. Figure 6 shows two faceted models M_1 and M_2, where nodes are labelled with the set they belong to (H, F, L), edges are labelled with the relation they represent ($feats$, tar, $facets$), and dashed edges correspond to the derived accessor methods $feats_H$ (for host objects) and tar_H (for links). The faceted models show to their right how they would be externally perceived. Model M_2 is a slice of the metamodel shown in Figure 1(a), recasted as a faceted model. Our formalization does not distinguish models (M_1) from metamodels (M_2) as both have the same structure. We do not consider inheritance relations as these can be emulated with relation $facets$. This way, two host objects can share the same facet, "inheriting" its content. Therefore, the facet notion generalizes inheritance as it works at both the class level and the object level. While facets and inheritance coexist in our implementation (see Section 8), future modelling frameworks

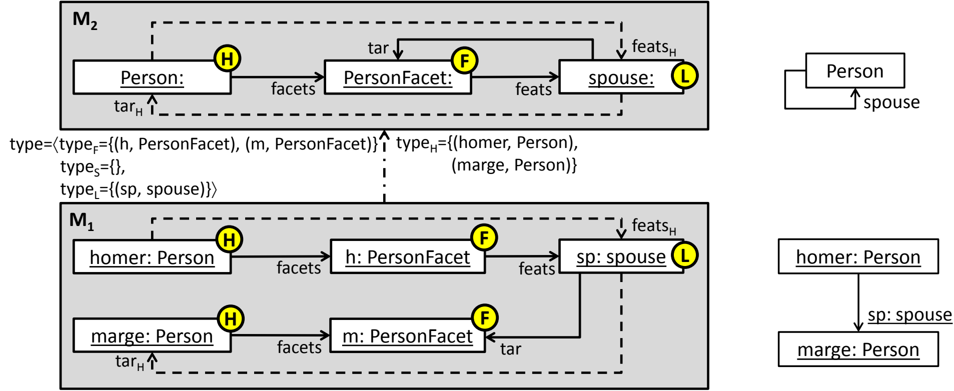


Fig. 6: Example of faceted models and faceted model type.

Next, we define the $type$ relation between two faceted models as three mappings between facets, slots and links. Different from standard modelling approaches, we do not type host objects, but their type is derived from that of their facets.

Definition 4.2 (*Faceted model type*). A faceted model type $type = \langle type_F, type_S, type_L \rangle \colon M_1 \to M_2$ between two faceted models $M_i = \langle H_i, F_i, S_i, L_i, feats_i, tar_i, facets_i \rangle$, for $i = \{1, 2\}$, is made of three partial functions $type_F \colon F_1 \nrightarrow F_2$, $type_S \colon S_1 \nrightarrow S_2$ and $type_L \colon L_1 \nrightarrow L_2$ s.t.:

$$
\begin{aligned}
\forall f \in F_1 \bullet type_F(f) \text{ is defined} \quad &\Longrightarrow \forall p \in feats_1(f) \bullet type_P(p) \in feats_2(type_F(f)) \wedge \\
&\qquad \forall f_2 \in facets_1(f) \bullet type_F(f_2) \in facets_2(type_F(f)) \\
\forall f \in F_1 \bullet type_F(f) \text{ is undefined} \quad &\Longrightarrow \forall p \in feats_1(f) \bullet type_P(p) \text{ is undefined} \wedge \\
&\qquad \forall f_2 \in facets_1(f) \bullet type_F(f_2) \text{ is undefined} \\
\forall l \in L_1 \bullet type_L(l) \text{ is defined} \quad &\Longrightarrow tar_2(type_L(l)) \in facets_2^*(type_F(tar_1(l)))
\end{aligned}
$$

(where we use $type_P$ for $type_S \cup type_L$).

The first two conditions in the definition demand compatibility of the type of a facet with the type of each of its features, and with the type of each of its assigned facets. The third condition demands compatibility of the type of the target of each link. Similar

to inheritance, this means that the type of a link target should be either the target facet of the link type, or a facet directly or indirectly assigned to this target facet. The definition uses partial functions to allow a model M to have typings w.r.t. zero or more other models M_i, where facets of M can be typed w.r.t. some of the M_i.

A faceted model type $type = \langle type_F, type_S, type_L \rangle \colon M_1 \rightarrow M_2$ induces a derived relation $type_H \subseteq H_1 \times H_2$ assigning types to host objects. The types of a host object are computed by collecting all facets assigned directly or indirectly to the host object, then retrieving the type of these facets using $type_F$, and finally obtaining the host objects holding those facet types. This can be formally defined as follows:

$$type_H \triangleq (({facets_2}^{-1})^+ \rhd H_2) \circ type_F \circ {facets_1}^+ \quad (3)$$

This derived relation allows objects to have no type (when they lack facets, or their facets have no type), one type (when the types of all facets they hold belong to exactly one host object), or multiple types (when the types of all facets they hold belong to more than one host object).

Example. Figure 6 shows a faceted model type example, where we use the object diagram notation to denote the types of facets, features and objects. Facets h and m in M_1 are typed by PersonFacet in M_2. The type of objects homer and marge, which is Person, is derived according to $type_H$.

Faceted models are a generalization of standard models in MDE. Figure 7 illustrates the extra possibilities that facet-oriented modelling brings. Model (a) shows that a host object can acquire facets whose type is held by different host objects. This makes the host object h to have two derived types through $type_H$: Person and Employee. We call this *multiple heterogeneous typing*. Model (b) is similar to model (a), but this time the host object holds two facets of the same type. We call this case *multiple homogeneous typing*. As mentioned in Section 4.1, this may produce an ambiguous access to the facet fields from the host object, which is solved by prefixing the field name with the facet name. Model (c) contains two host objects sharing a facet, which is similar to inheritance but agnostic on the metalevel: for models, it means that the host objects share the feature value, and for metamodels, it means that the host objects share the feature. Model (d) illustrates a more fine-grained control of sharing at the level of individual features. Model (e) shows that facets can have facets, which can be used to provide a hierarchical structuring mechanism for features.

The last model (f) is equivalent to a standard model in MDE. In standard models, objects have exactly one type and contain all their features, while features only belong to one object. This can be emulated by host objects having exactly one facet that cannot be shared and cannot be added or removed dynamically. This way, a standard model can be defined as a tuple $SM = \langle Ob, Pr, feats \subseteq Ob \times Pr \rangle$, where Ob is the set of objects, Pr is the set of features, and $feats$ assigns features to objects and is left-injective (no feature sharing) and surjective (no dangling features). The next lemma states that such standard models are a special case of faceted models.

LEMMA 4.3 (FACETED STANDARD MODEL). *A faceted model* $M = \langle H, F, S, L, feats, tar, facets \rangle$ *is equivalent to a standard model if:*

(*1*) $H \lhd facets$ *is a bijective function,*
(*2*) $F \lhd facets$ *is empty, and*
(*3*) $feats$ *is left-injective.*

PROOF. In appendix □

Example. The faceted models in Figures 6 and 7(f) are standard models, since $facets$ restricted to H is bijective. This means that each host object has exactly one facet, and

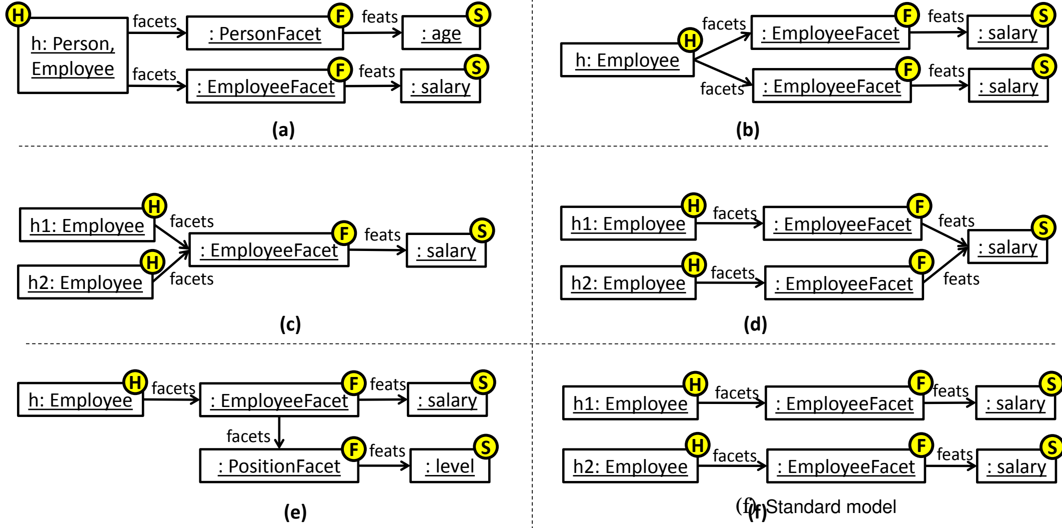


Fig. 7: Possibilities that facets bring to modelling.

each facet is assigned to exactly one host object. Moreover, the restriction of $facets$ to F is empty as no facet has another facet. Finally, $feats$ is left-injective as facets do not share any feature. In contrast, the faceted models in Figures 7(a-e) are not standard models because, in (a) and (b), the host objects have two facets (and so $facets$ is not a function); in (c), two host objects share a facet (and so $facets$ is non-injective and therefore not bijective); in (d), facets share a slot (and so $feats$ is not left-injective); and in (e), one of the facets has another facet (and so $F \lhd facets$ is not empty).

In MDE (see, e.g., [Ehrig et al. 2006; de Lara et al. 2007]) a standard model type between two standard models SM_1 and SM_2 is typically represented as a tuple of functions $type = \langle type_{Ob}, type_{Pr} \rangle \colon SM_1 \to SM_2$, with $type_{Ob} \colon Ob_1 \to Ob_2$ and $type_{Pr} \colon Pr_1 \to Pr_2$. These functions preserve the compatibility of objects and their features: $feats_2 \circ type_{Ob} = type_{Pr} \circ feats_1$, as the commuting diagram in Figure 8 shows.

$$
\begin{array}{ccc}
Ob_2 & \xrightarrow{feats_2} & Pr_2 \\
{\scriptstyle type_{Ob}} \uparrow & = & \uparrow {\scriptstyle type_{Pr}} \\
Ob_1 & \xrightarrow{feats_1} & Pr_1
\end{array}
$$

Fig. 8: Compatibility condition for standard model types.

Given a faceted model type $type = \langle type_F, type_S, type_L \rangle \colon M_1 \to M_2$, with M_i standard models, and $type_F$ a total function, we obtain a standard model type. This means that $type_H$ is a total function that commutes with $feats_H$, as required by Figure 8.

LEMMA 4.4 (STANDARD TYPE FOR STANDARD MODELS). *Given standard models M_1 and M_2, a faceted model type $type = \langle type_F, type_S, type_L \rangle \colon M_1 \to M_2$ is equivalent to a standard model type iff $type_F$ is a total function.*

PROOF. In appendix □

Altogether, we have seen how faceted models extend standard MDE models. Additionally, faceted models allow dynamic typing by considering the type $type_H$ of host objects a derived relation, and by enabling objects to acquire and drop facets. This acquisition and loss of facets should be consistent with Definition 4.1. Specifically, when a facet is no longer assigned to any object, it should be removed to keep relation $facets$ surjective.

Interestingly, it is possible to build a facet-oriented modelling framework on top of existing standard modelling frameworks, by enabling the use of objects as facets. This can be achieved by setting $H = \emptyset$ in Definition 4.1, providing a delegation mechanism to fetch the features from hosted facets, and considering multiple types for objects (as in the case of multiple heterogeneous typing). This setting has the practical advantage that there is no need to introduce new constructions, but regular objects can be used as facets. In the following, we assume this working scheme, which maximizes compatibility with existing modelling frameworks.

5. FACET MANAGEMENT

Next, we introduce some techniques for facet handling. In particular, Section 5.1 proposes a language to attach and detach facets from objects, Section 5.2 describes how to establish and maintain relations between features, and Section 5.3 shows different ways to extract views from faceted models.

5.1. Adding and removing facets

To simplify facet management, we have designed a language to manipulate facets. This relies on the use of queries to select the host objects to which facets should be attached or detached. The designed language is a domain-specific transformation language [Sánchez Cuadrado et al. 2014] (i.e., a constrained transformation language [Czarnecki and Helsen 2006]) with some singularities. First, we may need to refer to specific existing objects, and so, object identifiers may appear in queries. Second, any object can be used as a facet, as stated at the end of Section 4.2. Third, the facets may be existing objects, may be created uniquely for each selected host object, or may be created and shared by several host objects. Finally, it is possible to specify field value assignments and relations among fields that should be kept (e.g., equality of field values).

The language has two primitives to create and delete facets, whose structure is shown in lines 1–2 of Listing 1. Both primitives support four types of queries to select host objects (line 3): providing an object identifier, a collection of object identifiers, an OCL expression, or a pattern. The last three query types return a collection of objects. Primitive **addFacet** (line 1) adds a collection of facets to the selected objects. The facets can be existing objects (line 4) or new objects created by instantiating some class within a facet metamodel (line 5). In the former case, all selected host objects will share the existing facet; in the latter case, sharing the new facet among the selected host objects is done with the **reuse** option (see line 5), while omitting this option adds a new facet to each selected host. When the facet is a new object, it is possible to specify a value for its fields, or to define relations between its fields and the ones in the host object or its current facets. Section 5.2 will explain in more detail the handling of fields.

```
1 addFacet ⟨query⟩ ⟨facet⟩ (, ⟨facet⟩ )*
2 removeFacet ⟨query⟩ ( ⟨facetName⟩ | ⟨facetType⟩ )+
3 ⟨query⟩ ::= ⟨objId⟩ | ⟨objList⟩ | ⟨OCLexpression⟩ | ⟨pattern⟩
4 ⟨facet⟩ ::= ⟨facetName⟩: ⟨objId⟩ |
5             ⟨facetName⟩: ⟨facetType⟩ ( with { ⟨assignments⟩ } reuse? )?
```

Listing 1: Structure of the facet management primitives.

Primitive **removeFacet** (line 2) permits deleting facets from a set of host objects selected by a query. The facets to delete can be specified by name or by type. In the latter case, all

facets of the type are removed from the selected objects. If after applying this primitive, a facet is no longer assigned to any host object, the facet gets removed from the model.

As an example, Listing 2 adds a new facet of type Employee to object homer. Line 1 contains the object selection query, made just of the object identifier homer. We assume a unique way to identify objects. While this is native in our tool METADEPTH, metamodelling frameworks like EMF [Steinberg et al. 2008] may require selecting objects interactively by clicking on the user interface. Line 2 creates a new facet named emp of type Employee (a class from the metamodel Employment). Lines 5–7 assign a value to the facet fields. In addition, line 4 establishes an equality relation between the fields homer.fullName and emp.name. Removing the annotation "[**equality**]" from line 4 would copy the value of homer.fullName to emp.name, but the equality relation between both fields would not be maintained whenever either of them changes. Section 5.2 will provide more details on field assignment semantics.

```
1 addFacet homer
2     emp: Employment.Employee
3     with {
4         name = fullName [equality]
5         salary = 22345
6         ssNumber = 12345
7         active = true
8     }
```

Listing 2: Adding a new Employee facet to homer.

The excerpt in Listing 3 illustrates the use of OCL to define object selection queries. In this case, a different facet emp is added to every Person object with age bigger than or equal to 18. We use the symbol '$' to delimit the OCL expressions.

```
1 addFacet $ Person.allInstances()→select(p | p.age>=18) $
2     emp: Employment.Employee ...
```

Listing 3: Adding a new Employee facet to every adult.

While the previous query style facilitates selecting homogeneous objects (i.e., with same or compatible type), a pattern-like query may be more appropriate to select objects that are heterogeneous or need to satisfy complex relations. Hence, we support pattern-based selection using object tuples over which conditions can be specified using the keyword **where**. As an example, the query in Listing 4 selects every two Person objects, and if married, assigns a single facet named emp to each of them.

```
1 addFacet <h:Person, w:Person> where $ h.spouse=w $
2     emp: Employment.Employee ...
```

Listing 4: Adding a new Employee facet to every married Person.

Objects receive the types of the facets they hold, and hence, they may have multiple types. To select objects with multiple types, we support intersection types [Reynolds 1998] in patterns, specified as $\langle$obj$\rangle$: $\langle$Type$_1\rangle$ & ... & $\langle$Type$_n\rangle$. This returns the objects having all types Type$_1$, ...,Type$_n$, and the fields declared in those types can be used safely. Pattern declarations can also include type negation (!$\langle$Type$\rangle$) to demand that an object does not have the given type. Listing 5 shows an example where object h is required to have Person and Owner types, while w must be a Person but not an Employee.

```
1 addFacet <h:Person & Owner, w:Person & !Employee> where $ h.spouse=w $
2     emp: Employment.Employee ...
```

Listing 5: Using intersection types and type negation.

Objects can also be added more than one facet at a time. For example, Listing 6 creates two Employee facets for homer. Both facets share the fields ssNumber and active (lines 11-12), while their field name is synchronized with the field fullName of homer.

```
1  addFacet homer
2     dayJob: Employment.Employee with {
3        name = fullName [equality]
4        salary = 15000
5        ssNumber = 12345
6        active = true
7     }
8     nightJob: Employment.Employee with {
9        name = fullName [equality]
10       salary = 16400
11       ssNumber = dayJob.ssNumber [equality]
12       active = dayJob.active [equality]
13    }
```

Listing 6: Making homer a moonlighter.

Finally, several objects can share a facet. This can be done in three ways. The first one is to assign an existing facet to another host object. For example, by using **addFacet** marge emp: homer.dayJob, the object marge is added homer's facet dayJob (created in Listing 6) renamed as emp. The second is to use a pattern-like query, as all objects in every match of a pattern share the same facet. For example, in Listing 4, the two Person objects of each married couple share a same Employee facet. The third way is to use the keyword **reuse**, as this adds the same facet to all objects returned by a query. For example, adding **reuse** after the command in Listing 3 makes all adult Persons share the same facet, and adding **reuse** to Listing 4 makes all married Persons share the facet.

Regarding the requirements established in Section 2, the presented primitives permit the manual acquisition and loss of facets, covering **R2** and partially **R3**. Section 6.2 will propose an automated facet management mechanism to fully satisfy **R3**.

5.2. Field assignment semantics

This section details the two possible semantics for field assignment in primitive **addFacet**: *value* semantics and *reference* semantics. In the first case, a literal or the result of an (OCL) expression is assigned to a facet field at creation time. Lines 3 and 4 of Listing 7 illustrate value semantics. In line 3, a literal is assigned to ssNumber, while in line 4, the value resulting from the expression 100 * self.age is assigned to salary when the facet is created. The value of self.age (which corresponds to homer.age) is assigned when the **addFacet** primitive is executed. We use the "$" delimiter to denote value semantics, where the delimiters can be omitted for literals, so that ssNumber = 12345 is a shortcut for ssNumber = 12345.

```
1  addFacet homer
2     job: Employment.Employee with {
3        ssNumber = 12345              // value semantics: literals
4        salary = $ 100 * self.age $   // value semantics: expressions
5        name = fullName [equality]    // reference semantics: bi-directional synchronization
6        active = [self.age < 65]      // reference semantics: reactive field adapter
7     }
```

Listing 7: Different field assignment semantics.

In contrast, an assignment with reference semantics establishes the value of a field in reference to other fields, not only at creation time, but also later whenever the other fields change their value. Assignments with reference semantics can be arbitrary expressions delimited by "[" and "]", as in line 6 of Listing 7. We call these expressions *reactive field adapters*. In the example, whenever the value of homer.age changes, the adapter is evaluated and the result is assigned to field active. This assignment creates a unidirectional dependency: if age changes, then the value of active is updated, but active is not a derived attribute because its value can be changed manually and such changes are not propagated to age.

N-ary dependencies between fields of either facets or host objects are possible by defining several adapters. As an example, consider Listing 8 where we assume that class Person has a field rich. In line 3, the adapter salary = [100 * self.age] initially sets slot salary in the facet to 100 * self.age, and whenever homer.age changes, salary is updated using this expression. The subsequent adapter [rich = self.salary > 10000] sets homer.rich to true whenever salary in the facet changes to become bigger than 10000, and to false when salary becomes smaller than or equal to 10000. The adapter is within brackets since the assigned variable rich belongs to the host object homer and not to the facet.

```
1 addFacet homer
2     job: Employment.Employee with {
3         salary = [100 * self.age] [rich = self.salary > 10000]
4     }
```

Listing 8: Reactive field adapters for slots.

As this example shows, there may be dependencies between adapters. We say that a dependent set of adapters is ill-behaved if changing a field x triggers a sequence of updates that leads to having to update the field x again (with a different value), with the risk of creating an infinite cycle of updates. For example, name = [fullName] [fullName = 'Mr. '.concat(name)] is not well-behaved as setting fullName to 'Homer' triggers an update of name to 'Homer' due to the first adapter, which in turns causes an update of fullName to 'Mr. Homer' due to the second adapter, and this triggers again a modification of name due to the fist adapter. At run-time, we use a safe policy where each field changes at most once in a cycle, to avoid infinite cycles of changes. However, these situations may signal an error in the facet specification. In Section 7, we provide mechanisms to study if a cycle of field computations is well-behaved.

As demanding equality of field values is common, we have introduced the equality shortcut notation for the case of simple bi-directional (*bx*) synchronization adapters. This way, name = fullName [**equality**] is a shortcut for name = [fullName] [fullName = name]. This theoretically corresponds to feature sharing, as Section 4.2 described. Other bx synchronization relations between two fields need to be established explicitly with a pair of reactive adapters. For example, salary = [100 * self.age] [age = salary/100] updates the salary when the age changes, and vice versa. We leave for future work the possibility of an improved syntax, and mechanisms enabling the specification of one acausal relation between fields instead of two adapters.

Reactive adapters for links and slots behave similarly. As an example, the command in Listing 9 adds a Company facet to every Address object, and populates the employees link with all Employee objects living in the address. Theoretically, this link would be updated whenever the adapter yields a different value. In practice, for efficiency reasons, our implementation only performs the update if the adapter is *type-local*, meaning that it only uses fields of the object that acquires the facet (a in the example). Section 8 will provide more details about this. The adapters in Listings 7 and 8 are type-local. However,

the one in Listing 9 is not because it involves a global query (Person.allInstances()), and so, employees is assigned the result of the adapter when the facet is created, but this value is not updated automatically later. Hence, when adapters are not type-local, the reference semantics behaves as the value semantics.

```
1 addFacet <a:Address>
2     comp: Employment.Company with {
3         employees = [Person.allInstances()→select(p | p.isKindOf(Employee) and p.address=a)]
4     }
```

Listing 9: Reactive field adapter for a link.

Finally, the fields of shared facets can use the collection shortcut notation to assign them either all elements retrieved in the command query, or another facet in the same command. For example, Listing 10 adds a shared facet with type Company to all adult Person objects, and its field employees will contain all such persons matched by the command. Hence, employees = p [**collection]** is a shortcut for employees = [Person.allInstances()→select(p | p.age>=18)]. Alternatively, we could have used employees = job [**collection]** with the same result, as this would assign all facets named job, created in line 2, to employees. Section 9.2 provides more examples of this shortcut.

```
1 addFacet <p:Person> where $ p.age>=18 $
2     job: Employment.Employee with { ... }
3     comp: Employment.Company with {
4         employees = p [collection]
5     } reuse
```

Listing 10: Reactive adapter using the collection shortcut.

Overall, facet sharing and field adapters lead to a reactive approach to maintain consistency relations. While in standard MDE, one can set OCL constraints demanding some relation between fields across objects, there is no standard way to enforce those constraints. Our proposal is a novel way to do so.

With regards to the requirements in Section 2, our support for both predefined and user-defined relations between owned and acquired fields meets **R4**.

5.3. Model scenes

Once we enrich objects with facets, it is natural to have means to visualize faceted models according to several slicing criteria regarding facets. This way, we propose different manners to visualize a model with faceted objects. We say that they deliver a *scene* of a model, as they effectively look at a model under a certain angle, utilizing the facets as visualization criterion, similar to read-only non-materialized model views [Atkinson et al. 2015; Bruneliere et al. 2019]. Next, we present the three kinds of scenes we consider: total, sliced and granulated.

Total scene. The default visualization shows each host object reexposing the types and fields of its facets, as if the types and fields were owned. We call these scenes *total*. Figure 9(a) shows a total scene of the Springfield model, where homer has the two Employee facets created in Listing 6 (dayJob and nightJob). In the figure, fields show their facets to the right.

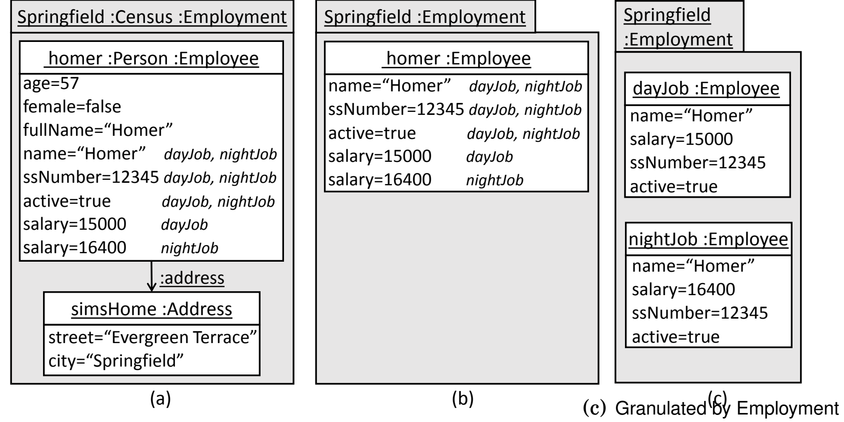


Fig. 9: Three *scenes* of the same model.

Sliced scene. Since a model may have objects with facets typed by different facet metamodels, we enable slicing a model with respect to a facet metamodel. Specifically, a *sliced scene* visualizes all objects of a model that are typed by a given facet metamodel. The slicing preserves the host objects, but it only shows the types and fields coming from the facet metamodel the model is sliced to, while objects without facets from this metamodel are omitted. Using model view terminology, a sliced scene is a projection of a model using a facet metamodel, where each host object displays its facets aggregated within the object. For example, Figure 9(b) shows a sliced scene of the Springfield model with respect to the Employment metamodel. homer is an object and not a facet, but it shows the type and fields from the Employment metamodel, and omits the type and fields from the Census metamodel. The scene does not show the simsHome object because it has no facet from the Employment metamodel. Sliced scenes are formally defined next.

Definition 5.1 (*Sliced scene*). Given a faceted model $M = \langle H, F, S, L, feats, tar, facets \rangle$ and a set $T = \{type^i \colon M \to M_i\}_{i \in I}$ of faceted model types, a sliced scene of M w.r.t. $type^j \in T$, written $SLICE(M, type^j) = \langle H', F', S', L', feats', tar', facets' \rangle$, is a faceted model defined as follows:

— $F' = \{f \in F \mid type^j_F(f) \text{ is defined}\}$,
— $H' = \{h \in H \mid \exists f \in facets^+(h) \cap F'\}$,
— $S' = \{s \in S \mid type^j_S(s) \text{ is defined}\}$,
— $L' = \{l \in L \mid type^j_L(l) \text{ is defined}\}$,
— $feats' = F' \lhd feats \rhd P'$, $tar' = L' \lhd tar \rhd F'$, $facets' = O' \lhd facets \rhd F'$.

Granulated scene. A *granulated scene* visualizes all facets of a model that are typed by a given facet metamodel. For this purpose, it reifies those facets as host objects, and then, it slices the model with respect to the facet metamodel. Using model view terminology, a granulated scene is a projection of a model using a facet metamodel, where each facet within a host object is displayed disaggregated. As an example, Figure 9(c) shows a granulated scene of Springfield with respect to the Employment metamodel, where the two facets of homer become visible as first-class objects, and homer itself is not visible. This way, the scene provides detailed employment data of Springfield. Granulated scenes are formally defined next.

Definition 5.2 (*Granulated scene*). Given a faceted model $M = \langle H, F, S, L, feats, tar, facets\rangle$ and a set $T = \{type^i\colon M \to M_i\}_{i\in I}$ of faceted model types, a granulated scene of M w.r.t. $type^j \in T$, written $GRAN(M, type^j) = \langle H', F', S', L', feats', tar', facets'\rangle$, is a faceted model defined as follows:

— $F' = \{f \in F \mid type^j_F(f) \ is \ defined\}$,
— $H' = \{(f, 0) \mid f \in F'\}$,
— $S' = \{s \in S \mid type^j_S(s) \ is \ defined\}$,
— $L' = \{l \in L \mid type^j_L(l) \ is \ defined\}$,
— $feats' = F' \lhd feats \rhd P'$, $tar' = L' \lhd tar \rhd F'$, $facets' = \{((f, 0), f) \mid f \in F'\}$.

Granulated scenes produce standard models, as each typed facet is promoted to a host object that contains exactly that facet, except for possible facet sharings.

Models can be filtered using OCL prior to scene generation. For example, to create a scene that only includes adult persons, we can define the filter Person.allInstances()→reject(p | p<18). Scenes are visualization devices, so Employee.allInstances() returns Set{homer} on model Springfield regardless of the scene. To account for facets, we extend OCL with the primitive ⟨type⟩.allFacets(). This way, Employee.allFacets() evaluated on Figure 9 returns Set{dayJob, nightJob}.

6. FACET LAWS AND INTERFACES

The previous section explained how to add facets to objects in an opportunistic way. In addition, we provide support for planned or systematic facet-based modelling. This is useful to control which elements can be used as facets, establish which facet types can be combined within a host object, or specify declarative conditions for acquiring or dropping facets so that they are enacted automatically. For this purpose, we propose *facet interfaces* and *facet laws*.

Figure 10 shows a scheme of these two mechanisms. It contains two metamodels, one playing the role of creation metamodel (CMM), and the other defining classes to be used as facet types (FMM). A facet interface for FMM specifies the classes of FMM that can be used to create facets (white boxes in the interface), identifies the classes that can be compatible facets of a same object, and may declare OCL constraints to be satisfied by any model that uses facets of FMM.

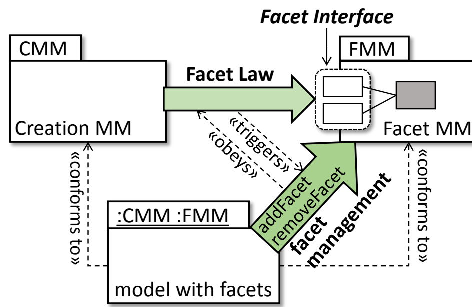


Fig. 10: Facet interfaces and laws.

Facet laws are specifications of how/when objects in the instances of CMM can acquire and drop facets from FMM. Laws can be used to check that **addFacet** commands issued

manually obey the laws, to complete new facets with default field values, and to trigger the acquisition and loss of facets automatically (which leads to fully satisfy the requirement **R3** specified in Section 2).

Our approach does not force the use of facet interfaces or laws, which remain optional. Next, Section 6.1 describes facet interfaces, while Section 6.2 introduces facet laws.

6.1. Facet interfaces

A facet interface restricts how a metamodel can be used for facet-based modelling. It declares the classes that can be used to create facets, their allowed combinations, and extra wellformedness constraints. The latter are useful to ensure additional requirements that arise when a same host object combines facets of different types, but which make no sense when the facet metamodel is used in a standard way.

Listing 11 shows the structure of a facet interface specification. It declares the public classes that can be used to create facets, either all or a list of them (line 2). It may define sets of compatible classes, i.e., classes whose instances can be facets of the same host object (line 3). Classes not defined compatible are incompatible. Finally, the interface may declare constraints that facets and their host objects should obey (line 4, where type is the context of the constraint and cName is the constraint name).

```
1 FacetInterface for ⟨MM⟩ {
2     public: all | ⟨typeList⟩
3     (compatible: ⟨typeList⟩ (, ⟨typeList⟩)*)?
4     (constraints: (⟨type⟩.⟨cName⟩ = ⟨OCLexpression⟩)+ )?
5 }
```

Listing 11: Structure of a facet interface.

Listing 12 shows an example of interface specification for the Employment metamodel. Line 2 indicates that all its classes can be used as facets. Line 3 states that host objects can take Employee and Owner facets simultaneously, and implicitly, precludes all other facet combinations. For example, objects cannot have facets of Employee and Company at the same time. Line 4 declares a constraint named repToIrreflexive in the context of class Employee, which forbids an employee reporting to itself. This cannot happen if Employment is used as a standard metamodel because classes Employee and Owner are disjoint, but the interface allows that possibility, so the constraint requires that reportsTo is irreflexive. Interestingly, the OCL expression seamlessly interprets that self is both an Employee (in self.reportsTo) and an Owner (in excludes(self)).

```
1 FacetInterface for Employment {
2     public: all
3     compatible: [Employee, Owner]
4     constraints: Employee.repToIrreflexive = $ self.reportsTo.excludes(self) $
5 }
```

Listing 12: An interface for using facets of Employment.

6.2. Facet laws

A facet law specification is a metamodel-level specification governing how instances of a creation metamodel can use facets of a facet metamodel. It is made of a set of facet laws which are mappings between classes in the creation and facet metamodels, to be either mandatorily or optionally satisfied by the instances of those classes. Each law can assign a default value to the facet fields, establish field relations, and define constraints. In addition, each facet law must respect the interface of the facet metamodel, if it exists.

Listing 13 shows the structure of facet law specifications. For generality, laws can apply to any number of creation and facet metamodels (see line 1). Hence, it is possible to specify laws for a metamodel that has facets from other metamodels; and to specify

```
1  FacetLaws for ⟨MM⟩ ( ( (, ⟨MM⟩)* with ⟨MM⟩ (, ⟨MM⟩)* ) | extensions ) {
2      ( (must | may) extend ⟨query⟩ with ⟨fctName⟩: ⟨fctType⟩
3          (with { ⟨assignments⟩ ⟨constraints⟩ } reuse? )? )+
4  }
```

Listing 13: Structure of a facet law specification.

laws for several creation metamodels. Section 9.3 will present an example of this. Moreover, using the keyword **extensions** permits applying a facet law to the subclasses of a given class, as we will illustrate in Section 9.4. The structure of mappings in a facet law (lines 2–3) is similar to the **addFacet** primitive, but in addition, they specify an optionality (*must* or *may*) and can add constraints to the facets.

Listing 14 shows a facet law specification governing Employment facets over Census models. Lines 2–3 specify that every adult person (age>17) must have an Employee facet. Alternatively, we could specify that a Person *may* have a facet Employee. If there is no law specifying that some objects (may or must) take a facet of a certain type, then it is forbidden. For example, Address objects cannot have any facet, and neither can Persons with age<=17. Lines 4–5 assign default values to the facet fields, and lines 6–7 define two constraints. The first one is a more restrictive constraint for the salary than the one in the Employment metamodel. The second one states that people over 65 cannot be active.

```
1  FacetLaws for Census with Employment {
2      must extend <p:Person> where $ p.age>17 $
3          with work:Employee with {
4              name = fullName [equality]
5              salary = 24000
6              minLocalSalary: $ self.salary>16000 $
7              retirement: $ self.age>65 implies not self.active $
8          }
9  }
```

Listing 14: Laws for Employment facets in Census models.

Facet laws are useful in the following scenarios:

- *To check manually issued facet management commands.* After issuing an **addFacet** or **removeFacet** command, every facet law that involves facet or creation types specified in the command is checked for consistency. This way, the command is not applied to the objects that do not satisfy the law condition (under-age persons in Listing 14).
- *To check faceted models for consistency.* Given a model with facets and a facet law, we can check if each object has the facets that the law establishes.
- *To complete addFacet commands.* A law establishes default values for facet fields (see lines 4–5 of Listing 14), so that they need not be repeated in every manual **addFacet** command. Thus, laws make knowledge on how facets should be created explicit.
- *To constrain facets.* By adding extra constraints to created facets (see lines 6–7 of Listing 14), facets are ensured to be consistent with their host objects.
- *To automate facet acquisition and loss.* Every mandatory (*must*) mapping in a law can be enforced automatically. For this purpose, an **addFacet** command is automatically issued whenever the mapping conditions are met, and a **removeFacet** is automatically issued when they no longer hold. For example, if we enforce the law in Listing 14, every adult Person object receives an Employee facet and two constraints (minLocalSalary and retirement). If we change the salary of an adult Person object to 14000, then the object violates the metamodel constraint minSalary and the law constraint minLocalSalary. If we decrease the age of an adult Person to 17, then the object automatically drops the

facet Employee. If we create a new Person with age 18, then it automatically acquires an Employee facet.

7. FACET ANALYSIS

This section presents two analysis techniques for facet laws based on model finding. The first one (described in Section 7.1) targets the analysis of their satisfiability, and the second one (shown in Section 7.2) analyses the well-behavedness of reactive adapters.

Both techniques are "push-button" analyses, fully automatic, so that the software engineer does not need to deal with the internal technicalities they involve. The first analysis returns a model demonstrating the applicability of the laws, and the second a model exemplifying an ill-behaved adapter, if such models exist. Both techniques are based on model finding using constraint solving [Jackson 2006]. This performs a bounded search up to models of a limited size. Therefore, our techniques are semi-decidable: finding a model demonstrates the analysed property, but the analysis result is inconclusive when the finder finds no model as some may exist outside the search scope. Anyhow, constraint solvers rely on the commonly agreed "small scope hypothesis" [Jackson and Damon 1996; Andoni et al. 2003], which assumes that most constraints are satisfied by models of limited size.

7.1. Facet law satisfiability

Our first analysis aims at answering whether a facet law specification is *applicable* to some instance of the creation metamodel. A facet law is not applicable if its query condition always returns no object regardless of the queried model, and is applicable otherwise. In addition, as lines 4–7 of Listing 14 illustrate, a facet law may add constraints to the objects to which it is applied, and initialize the facet fields. Hence, borrowing terminology from instantiability analysis of metamodels [Cabot et al. 2014], we say that a facet law is *satisfiable* if (1) it is applicable, and (2) the resulting model of some application satisfies the added constraints and the creation metamodel wellformedness constraints. We call a specification where at least one facet law is satisfiable *weak satisfiable*. We call a specification *strong satisfiable* if all its facet laws are satisfiable.

Example. The bottom of Figure 11 shows three facet laws defined over the simplified creation and facet metamodels on top. These are excerpts of the Census and Employment metamodels of Figure 1, where we have added a constraint diffNames requiring Persons to have distinct fullNames. The first law is not applicable because its query looks for pairs of Person objects with same fullName, but Persons have different fullNames due to constraint diffNames. The second law is applicable but not satisfiable because it assigns a salary that violates constraint minSalary. Finally, the third law is satisfiable because there are models where it can be applied (e.g., any model with an adult Person) and the result does not violate any constraint.

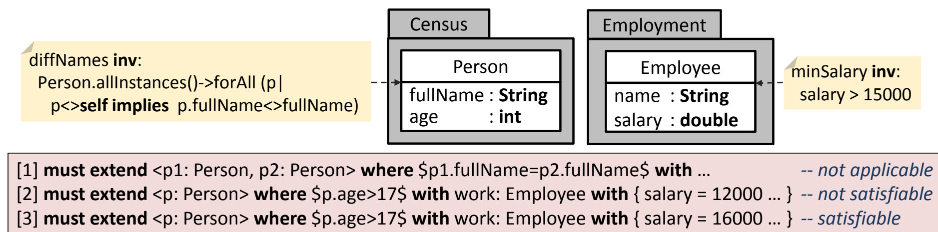


Fig. 11: Examples of facet laws: [1] not applicable; [2] applicable and not satisfiable; [3] satisfiable.

Our method to analyse law applicability and satisfiability is based on recasting facet laws as a constraint satisfaction problem and using model finders for the analysis [Jackson 2006; Kuhlmann and Gogolla 2012; Cabot et al. 2014]. An (OCL) model finder is a program that takes as input a metamodel with (OCL) integrity constraints, and outputs a valid instance model, if one exists within the search bounds. More in detail, our method builds a so-called *analysis metamodel* which "merges" the creation and facet metamodels, and is enriched with OCL constraints for representing the facet laws. Then, it uses a model finder to look for a valid instance of the analysis metamodel, in which case, the instance is a witness of the law applicability (or satisfiability).

Next, we explain in detail the steps in our analysis. The explanation will sometimes refer to Figure 12 as it summarizes the scheme to build the analysis metamodel and OCL constraints for a given facet law specification. As an illustration, we will use the example in Figure 13 to demonstrate how to analyse a facet law specification.

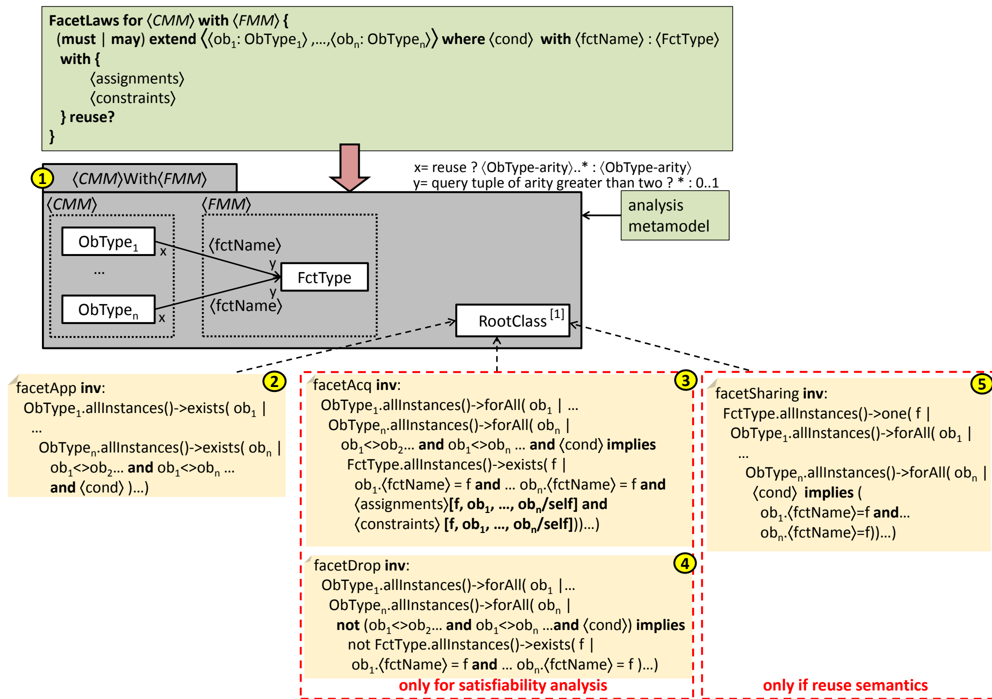


Fig. 12: Analysis scheme for satisfiability of facet laws.

(1) *Construction of the analysis metamodel*. In a first step, with label 1 in Figure 12, the method builds an analysis metamodel which contains the following classes:
   — all classes in the creation metamodels (⟨ObType$_i$⟩).
   — an auxiliary class RootClass that will hold the invariants generated from each facet law, as we will explain in step (2).
   — the class of all facets involved in the specification (⟨FctType⟩). The class of the facets that do not appear in any law are omitted, as they are unnecessary for the analysis. This reduces the analysis search space.

   In addition, for each facet law, the analysis metamodel includes a link from the class of the objects selected by the facet law to the mapped facet classes. Figure 12

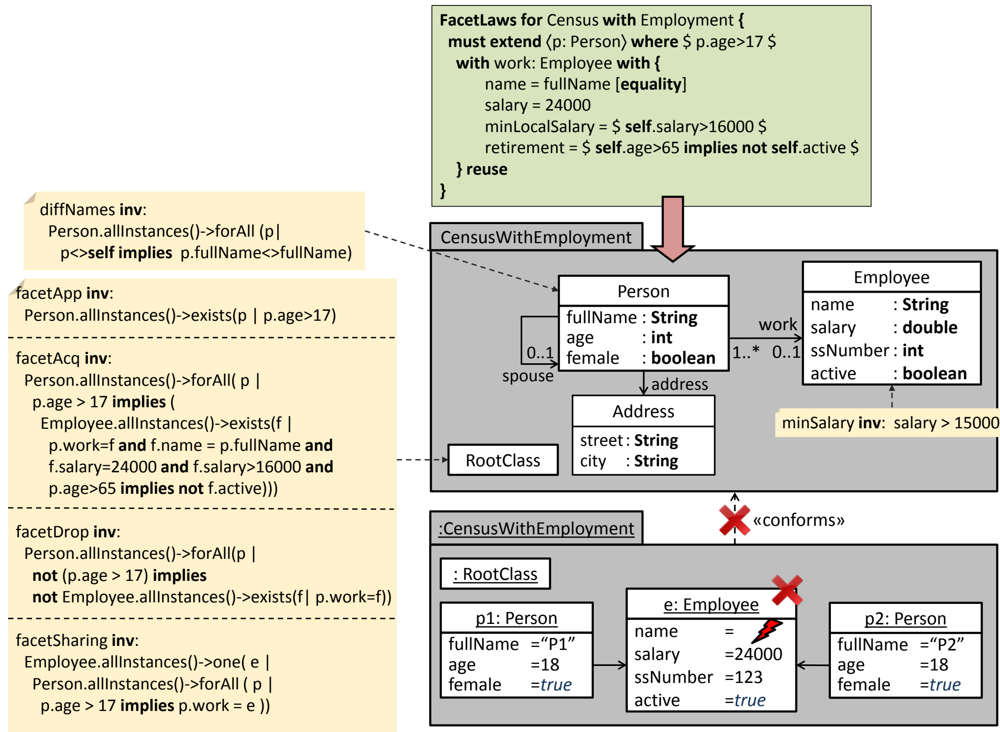


Fig. 13: Example of satisfiability analysis for a facet law specification.

assumes a facet law with a pattern-based query. In case of an OCL query, the class of the objects selected by the query needs to be identified in order to create the link from this class. For example, in the OCL query Person.allInstances()→select(...), the selected objects are of type Person.
The link cardinality to the side of the facet classes is set to optional, with unbounded maximum in case of a pattern-based query with arity bigger than one, and 1 otherwise. The reason is that in a pattern with more than one element, the same element in the model can be matched several times in different combinations. To the side of the creation classes, the minimum link cardinality is set to the arity of the given type in the pattern (i.e., the number of times the type appears in the pattern), and the maximum is set to * in case of reuse and to the minimum otherwise. The role name to the side of the facet class is the facet name ⟨fctName⟩.

Example. Figure 13 shows the analysis metamodel CensusWithEmployment, built from the facet law specification on top. This metamodel represents the facet relation between Person (creation class) and Employee (facet class) as a standard link (work). The cardinality to the side of Person is $1..*$ since Persons are selected individually and not in tuples (hence the 1), and the law has reuse semantics (hence the $*$). The cardinality to the side of Employee is $0..1$ since each Person is selected at most once by the query (i.e., the query is not based on patterns with arity bigger than one).

(2) *Encoding of the facet law semantics*. Next, the method adds to class RootClass three invariants (with labels 2–4 in Figure 12) for each facet law, plus an additional one (with label 5) in case the law has reuse semantics.
 — The invariant facetApp asks for a set of host objects satisfying the law query, i.e., it encodes the conditions for law applicability. If the query is pattern-based,

as is the case in the figure, then the invariant consists of nested exists clauses demanding the set of objects to be disjoint and to satisfy the condition ⟨cond⟩, similarly to [Guerra and Soeken 2015]. If the query includes an intersection type, then the invariant checks the existence of links from the creation to the facet types specified in the intersection type. For example, p: Person & Employee would be encoded as: Person.allInstances()→exists(p | p.work→notEmpty()). If the query includes a type negation (e.g., p: !Employee), then the invariant requires an object with no link to any facet of the given type.

— The invariant facetAcq expresses the constraints that a facet imposes to any host object that acquires the facet, i.e., it specifies the conditions for law satisfiability. The invariant selects the host objects satisfying the law query by using nested forAll clauses and checking the condition ⟨cond⟩. Then, it requires the existence of a facet connected to the selected objects, satisfying the conjunction of the assignments and constraints within the "with" clause of the law. In the law, the assignments and constraints are defined in the context of the facet, and so the invariant replaces "self" with f (the name of the selected facet object) or with the object name (ob_1, ..., ob_n) depending on the owner of the slot. If the maximum cardinality to the side of the facet is "*", then the invariant checks that the host objects have the facet using ob_i.⟨fctName⟩ →includes(f), instead of ob_i.⟨fctName⟩= f.

— The invariant facetDrop is the converse of facetAcq, and ensures that the host objects that do not satisfy the law conditions do not have an attached facet.

— The invariant facetSharing is created only when the law has reuse semantics. It ensures that all host objects satisfying the law condition share the same facet instance.

Example. Figure 13 shows the four invariants needed to analyse the satisfiability of the facet law specification on top. The invariant facetSharing is necessary because the law includes the keyword **reuse**. To analyse applicability but not satisfiability, facetApp would be the only generated invariant. Invariant facetAcq checks, among other things, that the constraint retirement introduced by the law is satisfied. For this purpose, the procedure substitutes self.age by p.age and self.active by f.active.

(3) *Model finding*. As a last step, the method resorts to a model finder to look for an instance of the analysis metamodel. The sought metamodel instance is demanded to contain exactly one RootClass object (represented in Figure 12 as a cardinality "[1]" in the top-right corner of the class) in order to ensure the non-vacuous satisfaction of its invariants. In addition, it must contain at least one instance of each creation and facet class in the facet law specification, and in case of reuse semantics, then it must contain at least two objects of the affected creation classes to effectively force facet sharing. If the model finder finds an instance of the analysis metamodel, then the applicability/satisfiability of the facet law specification is demonstrated; otherwise, the specification is not applicable/satisfiable within the search bounds.

Example. The facet law specification in Figure 13 has reuse semantics. Hence, the search configuration requires at least two objects of type Person, and at least one Employee. As the figure illustrates, the analysis metamodel is not instantiable within these search bounds because the facet requires name equality of each Person with the shared Employee facet, but invariant diffNames requires all Person names to be different. However, if we delete the keyword **reuse** in the facet law specification and repeat the analysis, then the search scope only requires at least one Person and one Employee. This search does return a model instance which demonstrates that the new specification is satisfiable within the new search bounds.

7.2. Well-behavedness of reactive adapters

As discussed in Section 5.2, facet laws may include expressions (adapters) triggering a change in one field as a reaction to changes in other fields. When there is a cyclic dependency in the adapter expressions, they may yield different values in each computation iteration. In that case, the adapters are *ill-behaved*. In practice, we do not update a field twice in a cycle, and so, infinite recursion is not possible. However, ill-behaved adapters may be a symptom of a mistake in a facet law, and therefore, we provide analysis support for their discovery.

The analysis is based on model finding. It extends the analysis presented in Section 7.1 as depicted in the scheme of Figure 14(a), which assumes the generation of an analysis metamodel as in Figure 12 but replacing invariant facetAcq by invariant facetAcq' (for simplicity, we assume just on creation class ObType).

When a field is assigned a dependent cycle of adapters, our new analysis: (i) creates a duplicate of the field that causes the cycle (obField' in Figure 14(a)); (ii) assigns to the duplicate field the result of the adapter that causes the cycle (last line of invariant facetAq'); and (iii) checks whether the duplicate and original fields can have different values (invariant illBehaved), as this would indicate that the adapter is ill-behaved.

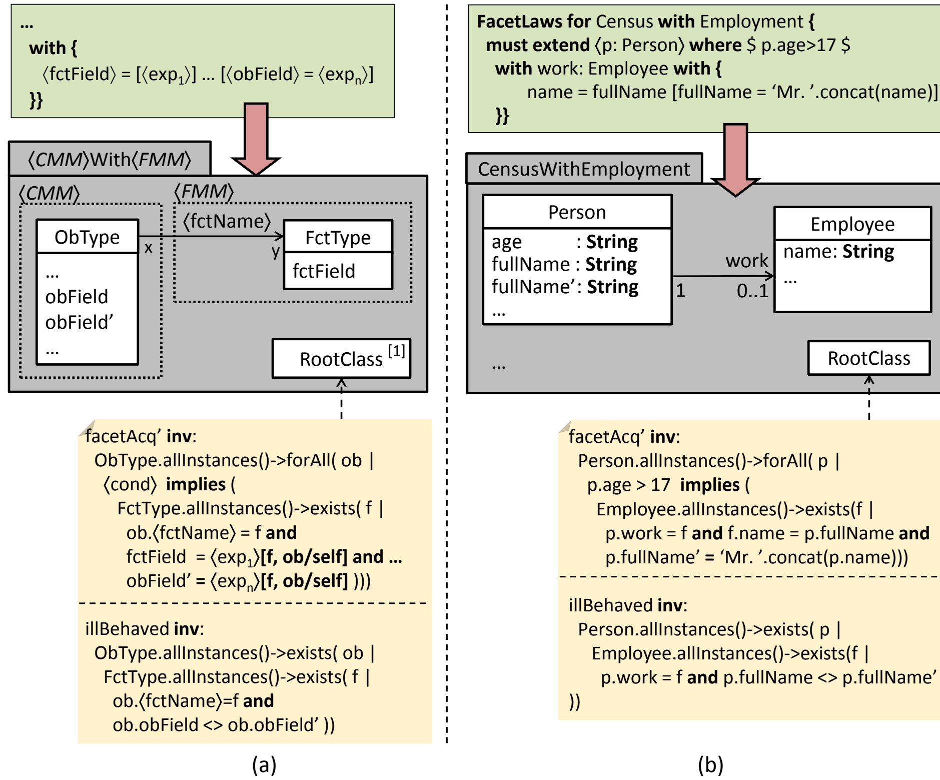


Fig. 14: Well-behavedness analysis of reactive adapters. (a) Scheme. (b) Example.

Figure 14(b) exemplifies the analysis for the field assignment name = [fullName] [fullName = 'Mr. '.concat(name)]. The two adapters in the assignment have a dependency cycle since the first adapter uses fullName to compute name, and the second adapter assigns a new value to fullName. To check whether the adapters are ill-behaved, our method creates

an analysis metamodel where: (i) class Person has an extra attribute fullName’; (ii) the invariant modelling facet acquisition (facetAcq’) uses fullName’ to set the value of the second adapter; and (iii) the invariant illBehaved checks whether it is possible that fullName and fullName’ have different values in Persons with an Employee facet.

Using a model finder to seek an instance of the analysis metamodel yields a non-empty solution (e.g., a model containing a Person with fullName=’Homer’ and fullName’=’Mr. Homer’). This indicates that the adapters are ill-behaved.

8. TOOL SUPPORT

We have implemented facets, facet interfaces, facets laws and their analyses on top of our tool METADEPTH [de Lara and Guerra 2010]. The tool and the examples in this paper are available at http://metadepth.org/mtl.

METADEPTH is a textual modelling tool that supports an arbitrary number of metalevels [Atkinson and Kühne 2001; de Lara et al. 2014]. Elements at every metalevel can be both classes (i.e., types of lower level elements) and objects (i.e., instances of upper level elements). For this reason, they are called *clabjects*, from the union of the words class and object. This makes METADEPTH level-agnostic [Atkinson et al. 2010a].

Listing 15 shows the definition of the Census metamodel in METADEPTH’s syntax, and Listing 16 has an excerpt of a Census model called Springfield. The keyword to declare a metamodel (more precisely, a model with no ontological type) is Model, and the one to create classes (i.e., clabjects with no ontological type) is Node. Their name can be used to create instances as in Listing 16, where line 1 creates an instance of Census called Springfield, and lines 2–8 create an instance of Person called homer.

```
1  Model Census {
2      Node Person {
3          fullName: String;
4          age: int;
5          female: boolean = true;
6          spouse: Person[0..1];
7          address: Address[1];
8      }
9      Node Address {
10         street: String;
11         city: String;
12     }
13 }
```

Listing 15: Census metamodel.

```
1  Census Springfield {
2      Person homer {
3          fullName= "Homer";
4          age= 50;
5          female= false;
6          spouse= marge;
7          address= simsHouse;
8      }
9      Address simsHouse {
10         street= "Evergreen Terrace";
11         city= "Springfield";
12     }
13 ... }
```

Listing 16: Springfield model.

We have extended METADEPTH to support the definition of facets as a special relationship between clabjects, to incorporate into host clabjects the types provided by their facets, and to make the access to facet fields from host clabjects transparent using delegation. Figure 15 shows a high-level excerpt of the design, where the dark classes are new additions.

In this design, Clabjects may own facets, which are simply other Clabjects accessible via class FacetRef. Clabject methods querying for owned fields (like method fields in the figure) have been extended to query the fields of owned facets as well, implementing a delegation mechanism for field access.

To implement the dynamic acquisition and loss of facets according to a facet law, we add to all clabjects mentioned in the law condition two constraints that we call *triggered* (class TriggeredConstraint). The first one checks the OCL condition stated in the law, and triggers the creation of the facet upon the condition satisfaction. The second one checks the negated OCL condition and triggers the facet deletion. While we have developed triggered constraints specifically for facets, their design is general (e.g., they hold a collection of actions instead of one). However, they currently need to be “type-local”,

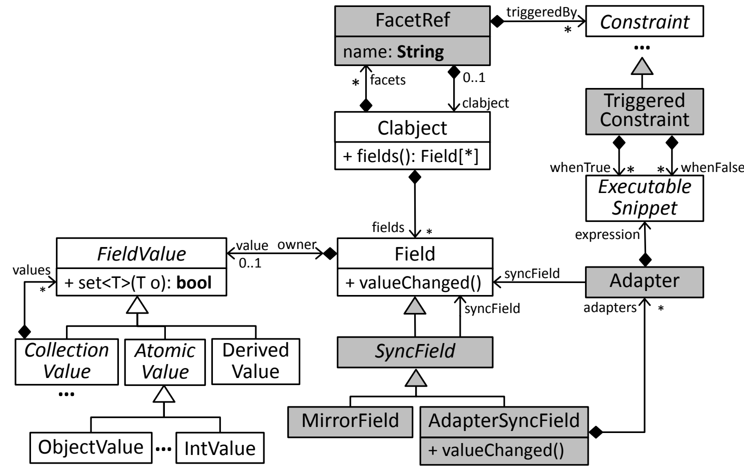


Fig. 15: Design of the realization of facets in METADEPTH (excerpt).

i.e., they are restricted to conditions on fields of the clabjects that acquire/drop the facet. For example, given the facet law in Listing 14, the condition p.age>17 in line 2 is type-local (p is of type Person, which is the type that acquires the facet), but the condition Address.allInstances()→exists(a | a.city='Springfield') is not. Non type-local conditions would require a static analysis to select the context clabject for the triggered constraint, or decide to place it in the model. This is left for future work.

METADEPTH featured a change notification mechanism, whereby changes in any field value (via method set in class FieldValue) are notified to the field owner (via method valueChanged in class Field). We have profited from this mechanism to implement reactive field adapters (class AdapterSyncField). These may need to trigger (push) recomputations to other fields, using an ExecutableSnippet. The equality relation between facet and host clabject fields is performed more efficiently by means of *mirror fields* (class MirrorField). These are field wrappers that allow sharing the value of a field with other field(s), so that there is only one FieldValue from which all fields in the relation read and set values.

In addition, we have extended the command set of METADEPTH to allow adding and deleting facets (**addFacet**, **removeFacet**), defining facet laws (**FacetLaws**), defining facet interfaces (**FacetInterface**), and enforcing facet laws (**addFacet all**). We have also extended the existing **dump** command – which produces a textual read-only representation, similar to Figure 16(b), of one or all the models currently loaded – to support scenes (cf. Section 5.3). This way, a command like dump using Employment shows the model sliced by Employment, while the command dump facets Employment shows a granulated scene using Employment. The command can also receive an OCL query, so that only the objects selected by the query are added to the scene. For example, dump using Employment $Person.allInstances()->select(p | p.female)$ only displays female Person objects in the scene.

METADEPTH integrates the Epsilon family of languages [Paige et al. 2009]. This permits defining in-place model transformations with EOL [Kolovos et al. 2006a], model-to-model transformations with ETL [Kolovos et al. 2008], and code generators with EGL [Rose et al. 2008]. Facets are transparent to all these languages, so they can be seamlessly used within Epsilon programs. For example, assume that we enforce the law in Listing 14, which adds an Employee facet to adult persons automatically. Then, the EOL program in Listing 17 causes the creation of a new facet Employee associated to p when line 2 is executed. Line 3 succeeds because p has now the field salary and can be accessed transparently. If we evaluate the correctness of the model at this moment via

```
1 var p : Person := new Person;
2 p.age := 23;                  // implicitly creates an Employee facet (as p.age > 17)
3 p.salary := 15100;            // OK, as p has now an Employee facet
4 p.age := 16;                  // p loses its Employee facet (as p.age <= 17)
5 p.salary := 21000;            // Error! p has no Employee facet
```

Listing 17: EOL program showing facet dynamicity.

the command **verify**, METADEPTH reports an error because the minLocalSalary constraint specified in the law (line 6 of Listing 14) fails. Line 4 makes the object p drop its facet Employee because it stops satisfying the law condition (i.e., the age is not greater than 17). Hence, line 5 causes an error due to an access to the undefined field salary.

Note that a static type checker may not be able to detect issues related to the dynamic nature of facets. For example, a static type checker would report an error in line 3 of Listing 17 because clabject Person lacks field salary (though a facet provides this field). Instead, we resort to dynamic type checking at run-time. The execution semantics of the language itself, be EOL or OCL, does not change though. This is so as the semantics of facets is handled by the modelling framework, which provides the information on the object types (which can be multiple and change at run-time) and fields (which may also change dynamically).

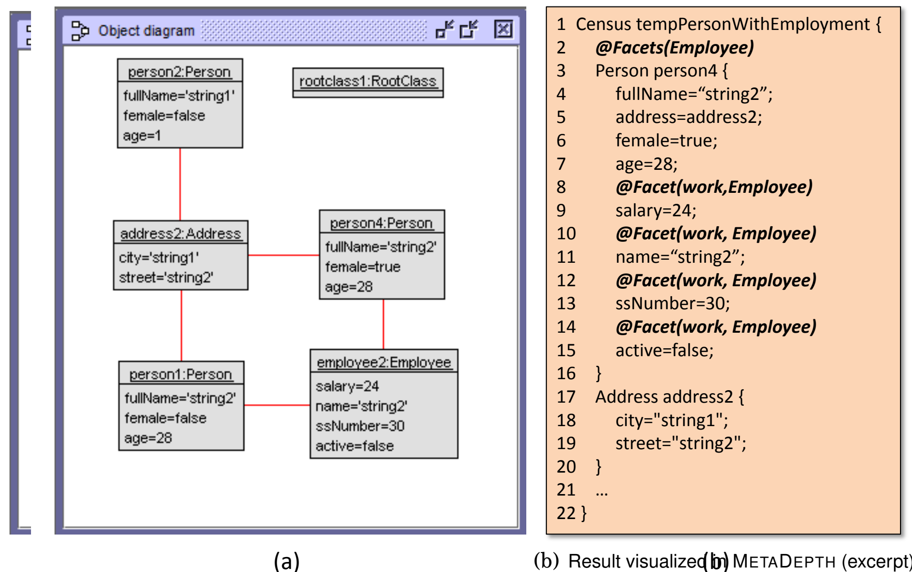
Finally, we have added to METADEPTH the new command **lawsat** that automates the analyses presented in Section 7. We rely on the USE validator as model finder [Kuhlmann and Gogolla 2012], which receives a class diagram with OCL constraints as input, and returns an instance of the class diagram if any exists within the given bounds. Internally, there are (Java) translators that transparently convert between the representations of USE and METADEPTH.

As an example, Figure 16(a) shows the result of the satisfiability analysis for the facet law in Listing 14, visualized within USE. The model illustrates the sharing of an Employee facet by every adult Person[2]. It is an instance of the synthesized analysis metamodel (cf. Section 7.1), and hence, it includes instances of the facet metamodel (like Employee) as well as a unique RootClass object. This model is parsed and represented back in METADEPTH as the faceted model in Figure 16(b). Facets are not shown as objects but embedded within their hosts as annotations [Sánchez Cuadrado and de Lara 2018]. This way, the annotation in line 2 states that object person4 has at least one facet of type Employee, and the annotations in lines 8, 10, 12 and 14 identify the slots of the work facet. The faceted model can be visualized in several ways using *scenes*, e.g., slicing by Employment.

9. CASE STUDIES

To evaluate the usefulness of facets, we now present five illustrative scenarios: handling of annotation models (Section 9.1), reuse of model management operations (Section 9.2), multi-view modelling (Section 9.3), multi-level modelling (Section 9.4), and language product lines (Section 9.5). Our goal is to showcase the benefits of our proposal, and to compare with other techniques for resolving the same scenarios. The selection of techniques is based on literature search, personal knowledge and availability of a tool. The comparison is based on requirements **R1**–**R4** listed in Section 2 and on more specific criteria for each scenario. The section concludes with a summary of this qualitative evaluation, and discussing strengths and limitations of our approach (Section 9.6).

[2]To reduce the search space, we have represented salaries in thousands of euros.



(a) (b) Result visualized in METADEPTH (excerpt)

Fig. 16: Satisfiability analysis.

9.1. Integrating annotation models

Several scenarios require annotating a domain model with additional information, e.g., regarding concrete syntax [Guerra and de Lara 2007], uncertainty [Famelis and Chechik 2019], access control [Bergmann et al. 2016], configuration of abstraction operations [de Lara et al. 2013] or variability [Salay et al. 2014]. This extra information can be provided as a separate annotation model that references the domain model. Then, services defined on the annotation model (e.g., reasoning services for uncertainty annotations [Famelis and Chechik 2019]) become available for the domain model.

Facets can be used for this purpose, which in addition provides dynamicity and a seamless connection between the annotated and the annotation model (since there is no need to explicitly navigate from the annotated objects to the annotations to access their fields). As an example, we show how to use facets to provide a graphical concrete syntax to domain models. The metamodel in Listing 18 defines simple graphical representations to be used as facets, like rectangles and circles, as subclasses of Element. These have a position (x, y, in lines 4–5), background and line colours (lines 6–7), a label (line 8) and can be connected (line 9).

A facet interface, shown in Listing 19, exposes Rectangle and Circle and makes them compatible, so that objects can be represented by none, either of them, or both.

Finally, Listing 20 contains the laws for adding ConcreteSyntax facets to Census models. They add Circle facets to female Person objects, and Rectangle facets to males. In both cases, the label in the concrete syntax is the person name, persons are linked to their spouse, and the size (radius or height/weight) is given by the age. Since the laws have reactive field adapters, changing the age or name of a Person updates the visual representation, changing the graphical size modifies the age, and changing the label modifies the name.

As a proof of concept, we have implemented a new METADEPTH command, called **draw**, which creates a visualization like the one to the left of Figure 17, from a ConcreteSyntax model. ConcreteSyntax models are made of facets over objects of another model, like Springfield. This provides a textual/graphical synchronization for free, as

```
1  Model ConcreteSyntax {
2      enum Color { red, green, blue, black, white, lightgrey }
3      abstract Node Element {
4          x: int;
5          y: int;
6          bckColor: Color = white;
7          lineColor: Color = black;
8          label: String;
9          linkedTo: Element[*];
10     }
11     Node Rectangle : Element {
12         width: int;
13         height: int;
14     }
15     Node Circle : Element {
16         radius: int;
17     }
18 }
```

Listing 18: Metamodel for graphical concrete syntax.

```
1  FacetInterface for ConcreteSyntax {
2      public : all
3      compatible : [Circle, Rectangle]
4  }
```

Listing 19: Facet interface for graphical concrete syntax.

```
1  FacetLaws for Census with ConcreteSyntax {
2      must extend <p:Person> where $ p.female $
3          with c:Circle with {
4              label = fullName [equality]
5              linkedTo = spouse [equality]
6              radius = [2*age] [age = radius/2]
7          }
8      must extend <p:Person> where $ not p.female $
9          with r:Rectangle with {
10             label = fullName [equality]
11             linkedTo = spouse [equality]
12             height = age [equality]
13             width = age [equality]
14         }
15 }
```

Listing 20: Laws to add graphical concrete syntax to Census models.

for example, the graphical view is updated when a new Person is created textually, the graphical representation of a Person object changes from a circle to a rectangle (or vice versa) when the value of its female field changes, and the fields of Person objects are modified whenever some graphical field (like the radius) changes. Facets provide this synchronization natively, which otherwise need to be implemented ad hoc.

9.1.1. Comparison with other model extension approaches. Next, we compare several ways to extend a domain model with an annotation model, based on requirements **R1–R4** and the following specific criteria:

— *Dynamicity:* whether the approach permits adding/deleting annotations to/from domain objects at runtime, as well as annotating new objects automatically.
— *Sharing:* whether a domain object can have several annotations, or an annotation can be shared by several objects.

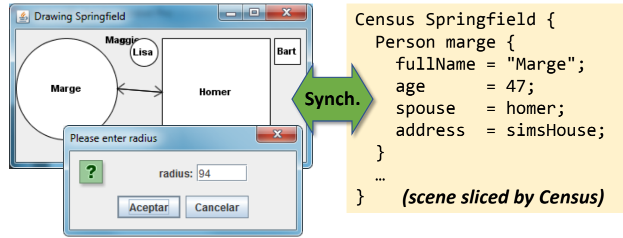


Fig. 17: Concrete syntax of the Springfield model via facets.

— *Field access:* whether accessing the fields of an annotation from a domain object (or vice versa) is done explicitly (e.g., navigating a link) or transparently.
— *Typing:* if the domain objects can be recovered using the type of the annotations.
— *Bi-directional and reactive change propagation:* whether the domain objects can get updated upon annotation changes, and vice versa (i.e., in a bi-directional way, *bx* in short). We only consider that this criterion is fulfilled when the support is native.

Dynamicity and bx change simplify the coordination of domain and annotation models; otherwise, workarounds like re-generation of the annotation model upon domain model changes may be necessary. Sharing enhances expressivity; otherwise, a domain object cannot have two annotations, or vice versa. Support for typing and field access simplifies the definition of transformations over annotated models; if unsupported, a transformation language able to cope with several input models (the annotated and the annotating models) is required, and the transformation becomes more complex due to the explicit navigation from the objects to their annotations and vice versa.

Table II compares facets with other approaches to solve this scenario using both the specific criteria listed above and the requirements listed in Section 2. Cross-referencing is the baseline. It consists in having a separate annotation model that refers to the objects in the domain model, and vice versa, similarly to Figure 2(b). This approach is not dynamic, as creating annotations automatically for new domain objects is possible, but requires manual encoding. The approach allows any kind of sharing. However, field access is not transparent as one needs to navigate from the object to the annotation, and domain objects cannot be retrieved using the type of the annotations. Finally, there is no native support for bx change propagation.

Table II: Comparison of approaches for model annotation.

	Specific criteria					General criteria			
Approach	**Dynamic**	**Sharing**	**Field access**	**Typing**	**Bx change**	**R1**	**R2**	**R3**	**R4**
Cross-references (baseline)	no	yes	navigation	no	no	×	×	✓	×
EMF profiles [Langer et al. 2012]	limited[a]	yes	navigation	no	no	✓	×	∼	×
Decorator models [Kolovos et al. 2010]	no	limited[b]	transparent	no	no	✓	∼	✓	×
A-posteriori [de Lara and Guerra 2017]	limited[c]	limited[d]	transparent	yes	limited[e]	✓	✓	∼	×
Facets	yes	yes	transparent	yes	reactive adapters	✓	✓	✓	✓

[a] No mechanism to automatically attach and drop stereotypes. [b] Non-resolvable ambiguous access errors may occur when several annotation objects decorate the same domain object. [c] Annotation types cannot be assigned to objects manually, but automatically based on a specification. [d] Annotations cannot be shared among domain objects (but these may have several annotations). [e] Annotation objects cannot have state, but this has to be mapped to attributes of domain objects.

Stereotypes can be used to emulate annotations. This approach would add a profile (the annotation metamodel) to the domain metamodel. Dynamicity is limited, since there is no mechanism to automatically attach and drop stereotypes, but they have to be manually handled. Several stereotypes can be added to the same domain object, and stereotypes can be shared. Field access is not transparent as the stereotype and the domain object are different objects. Domain objects cannot be directly retrieved

from annotation types. Bx change propagation is not supported as the fields in classes and stereotypes cannot be synchronized natively. While profiles were proposed to extend the UML metamodel [UML 2017], some proposals have adapted them to EMF metamodels [Langer et al. 2012]. The underlying mechanism of EMF profiles consists of models with cross-references, having benefits over the baseline in terms of the specification of the correspondences and tooling.

Decorator models are proposed in [Kolovos et al. 2010] as a means to decouple a model from its annotations (e.g., performance annotations on an activity diagram). Decorations have their own metamodel and are persisted separately, cross-referencing the domain model. A transparent access to decoration fields from the domain model objects is possible using Epsilon's connectivity layer [Paige et al. 2009]. However, the approach does not allow bx synchronization of decorator and host object fields, multiple typing, or dynamicity of decorations based on conditions. Decorator sharing is limited as non-resolvable ambiguous access errors occur if several decorator objects of the same type decorate the same host object.

A-posteriori typing [de Lara and Guerra 2017] permits typing a domain model with respect to the types of an annotation metamodel. Dynamicity is limited because the new typing is automatic based on a specification, but new annotation types cannot be assigned to objects manually. Sharing is limited too, since annotations cannot be shared between objects. Field access is transparent, but annotations cannot add new fields to objects, i.e., all state must reside in the domain model. Domain objects receive the type of the annotation objects, and so these can be retrieved using that typing. Bx change propagation is limited as all state must reside in the domain model.

With respect to facets, they are dynamic, can be shared among host objects, and host objects can have several facets. Field access is transparent, domain objects can be retrieved by the annotation types, and bx change is supported natively.

Regarding requirements **R1**–**R4**, all approaches but cross-referencing are modular and non-intrusive (**R1**). Cross-referencing typically implies modifying the domain metamodel to add navigation links to the annotation metamodel, as otherwise, obtaining the annotations of a domain object would be inefficient (see footnote 1 in Section 2). Requirement **R2** asks for the possibility of objects to acquire new types, fields and constraints, and their transparent access. Using cross-referencing and profiles, accessing fields of annotations is not transparent but requires explicit navigation. Decorator models make this access transparent but objects cannot acquire new types (hence the $\sim$ in the table). **R3** demands both automatic and manual acquisition/loss of types, fields and constraints. EMF profiles do not provide automatic mechanisms for attaching/dropping stereotypes, while a-posteriori typing does not support manual acquisition of a-posteriori types. As for **R4**, only facets support synchronization between fields natively.

Coming back to our example of graphical/textual syntax synchronization, its implementation using cross-references would need to provide ad hoc support for bx. Table III shows the artefacts that need to be created or modified for adding the required bx support using either cross-references or facets. In the first case, we base on a full implementation developed within METADEPTH using cross-references and the Epsilon languages[3]. First, we had to modify the Census and ConcreteSyntax metamodels to include a reference to the Element a Person is represented by, and the object that an Element represents. Another option would be creating a new metamodel pointing to the other two, as in Figure 2(b), but this would complicate model synchronization. In addition, we implemented an ETL transformation that creates a graphical model from a Census model, and has to be manually re-executed when the Census model changes; and an EOL

[3]Available at http://metadepth.org/mtl/eval.

program to update the Census model upon changes in the graphical model. Finally, we modified the **draw** command to execute the EOL program upon changes in the graphical model. To synchronize models conformant to other metamodels than Census, we would need other transformations and some mechanism to identify the one to use in each case. Overall, the solution based on cross-references required modifying existing artefacts (metamodels, METADEPTH's Java code), creating new ones on four technologies (metamodelling language, Java, ETL, EOL), and some tasks are still manual (updating the graphical model). In contrast, in the solution using facets, we just need to write a facet law, and optionally a facet interface. The latter would be written by the designer of the ConcreteSyntax metamodel, and not by a developer that wants to add a concrete syntax to his/her language.

Table III: Artefacts required for implementing the graphical/textual syntax synchronization example (see Figure 17) using cross-references and facets.

Approach	Modified/required artefact	Purpose	Size (LOC)
Cross-references	• Modified census and graphical metamodels	• Add cross-references	2
	• New model-to-model transformation (ETL)	• Create graphical model from census model, manual re-execution upon census model changes	40
	• New program (EOL)	• Update census model upon graphical model changes	10
	• Modified **draw** command (Java)	• Re-execute EOL program if the graphical model changes	8
Facets	• New facet law	• Automatic consistency of census and graphical models	15
	• New facet interface (**optional**)	• Guide developers in creating facet law	4

Regarding the other approaches, a-posteriori typing is not fully applicable as it does not permit mapping facet fields like x or y onto the host object fields. Decorator models permit transparent field access, but do not allow bx field synchronization or changing the decorators automatically depending on conditions of the source model (e.g., changing the female field value). Stereotype-based approaches do not support conditional graphical styles (e.g., based on the female field). Overall, existing approaches either should be augmented with manually written code, or do not fully cover this scenario. Hence, we conclude that facets have advantages to tackle this scenario, and may simplify works like [Bergmann et al. 2016; Famelis and Chechik 2019; Guerra and de Lara 2007; de Lara et al. 2013; Salay et al. 2014].

9.2. Reuse of model management operations

Models are manipulated using model management operations, like model transformations and code generators. Such operations are defined over a metamodel and cannot be directly reused for a different one [Bruel et al. 2020]. Without proper reuse support, developers that want to reuse an existing operation typically clone the operation and adapt it for the new metamodel manually. Next, we show that facets can provide support for reuse that avoids this potentially costly and error-prone manual adaptation.

As an example, Listing 21 shows a simple metamodel to represent (not necessarily disjoint) groups of elements having an integer field. The goal of the metamodel is being able to generically define different metrics. Listing 22 contains an EOL operation defined over the class Group that computes the average of a group of elements.

```
1 Model Metrics {
2     Node Group { elems : Element[*]; }
3     Node Element { quantity : int; }
4 }
```

Listing 21: Metrics metamodel.

```
1 operation Group average () : Real {
2     if (elems.isEmpty()) return 0;
3     return elems.collect(quantity).sum()/elems.size();
4 }
```

Listing 22: Metrics operation.

We would like to reuse the operations defined over the Metrics metamodel to measure Census models. With our approach, this can be done by adding Element facets to the set of Person objects of interest, and mapping Element.quantity to the property to be measured. Listing 23 shows a facet law specification that creates a group with all adult Persons, mapping Element.quantity to Person.age. All adult Persons share a Group facet (see **reuse** in line 8), and have a different Element facet that belongs to the field elems of the group. By using [**collection**] in line 7 (cf. Section 5.2), every time there is a new adult Person, it is automatically added a new Element facet, which in turn is added to field elems.

```
1 FacetLaws for People with Metrics {
2     must extend <p:Person> where $ p.age>17 $ with
3         ageMetric: Element with {
4             quantity = age [equality]
5         },
6         averageAge: Group with {
7             elems = ageMetric [collection]
8         } reuse
9 }
```

Listing 23: Laws to make Census models measurable.

After defining the laws, we can execute the EOL program in Listing 24 over any Census model to compute the average of all objects with facet Group, without the need to adapt the original operation in Listing 22. In this case, it returns the average age of all adults. Moreover, if the age of a Person changes, the corresponding field quantity changes as well. If a new adult Person is created, the appropriate Group and Element facets are automatically added/deleted.

```
1 operation main() {
2     for ( m in Group.allFacets() )
3         ('Average '+m.average()).println();
4 }
```

Listing 24: Reusing operation average for Census.

9.2.1. Comparison with other reuse approaches. Next, we compare with other approaches to reusing model operations and transformations across metamodels. We base the comparison on **R1**–**R4** and on the following subset of criteria identified in [Bruel et al. 2020]:

— *Reuse interface:* this is the exposed reuse interface of the reusable operation.
— *Checking type:* whether it is possible to validate that an operation is being correctly reused, either enabling static checking (simple type-checking) or semantic checking (e.g., wellformedness constraints).
— *Style:* whether the objects over which an operation is to be reused are specified by extension (i.e., enumerating them) or by intension (i.e., providing selection conditions).
— *Multiple occurrences:* whether it is possible to define multiple reuse contexts for an operation within a metamodel.
— *Adaptation:* whether there are means to bridge the heterogeneities between the metamodel over which an operation is defined, and the metamodel where it is reused.

Table IV compares several reuse approaches according to the specific criteria listed above and the requirements of Section 2. We take model adaptation as the baseline. This consists in translating the models to be manipulated into instances of the metamodel over which the reused operation is defined. In our example, it means transforming the Census models into Metrics models. Thus, reuse is achieved by means of a transformation, and so, any adaptation of the source model is possible by using an intensional style. However, there is no specific checking of the transformation correctness (thorough

testing is required). Multiple occurrences are allowed; e.g., to measure the average age and weight of Persons, these can be transformed into Elements within two different Groups. However, this solution is heavyweight and complicates management, because when the Census model changes, we need to re-execute the adaptation transformation before running the metrics operation. Furthermore, if the reused operation makes in-place modifications, then another transformation is needed to propagate the changes backwards.

Table IV: Comparison of approaches for model management operation reuse.

	Specific criteria					General criteria			
Approach	**Reuse interface**	**Checking type**	**Style**	**Mult. occur.**	**Adaptation**	**R1**	**R2**	**R3**	**R4**
Model adaptation (baseline)	MM	no	intension	yes	arbitrary	✓	✓	×	∼
A-posteriori [de Lara and Guerra 2017]	MM	syntactic	extension intension	no	arbitrary, bx, derived feats	✓	✓	∼	×
Concepts [Sánchez Cuadrado et al. 2014]	MM	syntactic	extension	no	arbitrary, derived feats/classes	✓	✓	∼	✓
Model typing [Guy et al. 2012]	MM	syntactic semantic	extension	no	renaming, derived feats	✓	✓	∼	✓
Facets	MM, interface	syntactic semantic	extension intension	yes	arbitrary, bx	✓	✓	✓	✓

A-posteriori typing can assign types of the Metrics metamodel to the Census models, so that the metrics operations can be applied directly on them. Compared to the baseline, this approach is bx as the operation is executed directly in the Census models, but considering the assigned Metrics typing.

Concepts [Sánchez Cuadrado et al. 2014] is a reuse approach inspired by generic programming. Model operations are defined over concepts (simple metamodels), which need to be bound to a concrete metamodel in order to reuse the operation. This binding induces a high-order transformation that rewrites the operation to make it applicable to the metamodel.

Model typing [Guy et al. 2012] requires specifying a subtyping relation between the metamodel where an operation wants to be reused, and the metamodel for which the operation is defined. This enables a safe reuse of the operation over instances of the former metamodel. Differently from the previous approaches, it supports pre/postcondition *semantic* checks.

Facets can be seen as an extension of a-posteriori typing with increased control of the reuse interface by facet interfaces; support for semantic checking expressed as OCL constraints; more expressive mappings through sharing and reactive field adapters; and support for multiple occurrences of the reused operation (e.g., to compute metrics by several criteria). In particular, the latter has been identified as a lacking feature in current approaches to transformation reuse [Bruel et al. 2020].

Regarding the requirements **R1**–**R4**, all approaches are modular and non-intrusive (**R1**) and can emulate the acquisition of types, fields and constraints by objects (**R2**). Concerning **R3**, all approaches provide some automated mechanism for the acquisition of types, fields and constraints (a transformation in model adaptation, a retyping specification in a-posteriori typing, a binding in concepts, a subtyping specification in model typing, and laws in facets). However, in model adaptation, the acquired types, fields and constraints are not updated automatically upon model changes, but the transformation needs to be re-executed. In contrast, this update is automatic in the other approaches as they do not rely on a pre-processing of the manipulated model. Regarding the manual selection of individual objects, it is only supported by facets. Finally, with respect to **R4**, model adaptation permits arbitrary relations between the fields of the original and adapted models, but restoring these relations upon model changes is not automatic (hence the $\sim$ in the table). Instead, restoring these relations

```
1 Model MedicalRecords {
2     Node Patient {
3         name : String;
4         insurance : boolean;
5         surgeries : Surgery[*];
6     }
7     Node Surgery { desc: String; }
8 }
```

Listing 25: Facet metamodel for health.

is automatic in concepts, model typing and facets, while in a-posteriori typing acquired fields are read-only.

To deal with the example in Listing 24, model adaptation requires transforming the Census models into Metrics models, and reapplying the transformation whenever the Census models change. A-posteriori typing and model typing cannot directly deal with Group objects as the Census metamodel lacks an equivalent class. While concepts can tackle Group objects by defining a derived class, the approach is currently only applicable to the ATLAS Transformation Language (ATL) [Jouault et al. 2008].

9.3. Multi-view modelling

Frequently, complex systems are modelled by separate descriptions in different views. This permits separation of concerns and helps tackling the system complexity [Atkinson et al. 2015; Bruneliere et al. 2019]. While each view is understandable on its own, to achieve a meaningful description of the system, views need to be consistent with each other. Facets, scenes and laws can support multi-view modelling, as we describe next.

Building on the running example, assume we would like to describe a system made of three types of views: a Census view, an Employment view and a MedicalRecords view. Each view is specified according to its metamodel, while facet laws describe how they relate to each other. The metamodels for Census and Employment were presented in Figure 1. Listing 25 shows an excerpt of the metamodel for MedicalRecords, where Patients may have an insurance and go through surgeries.

Facet laws can be used to specify how different view types are related and can overlap. For example, the laws in Listing 14 describe how a Census view is related to the Employment view, and in particular, they overlap in classes Person and Employee under certain conditions. In addition, facet laws also permit specifying constraints involving several views, as Listing 26 shows. These laws specify the relation of the Census and Employment views with MedicalRecords. The law in lines 2–6 specifies that any Person that is an Employee (i.e., has a facet Employee) needs to have a medical record (i.e., it must have a Patient facet). To specify that the object has both types Person and Employee, we use an intersection type (cf. Section 5.1). The law in lines 7–12 states that Persons who are not Employees (i.e., Person & !Employee) may have a private insurance. Finally, the law in lines 13–17 specifies that spouses of Employees have a medical record and get the insurance of their partner.

Once view types and their relations are defined, views are given by models. Overlapping in view types is realized by objects with facets. Views can be separated in different models, as Figure 18 shows. The figure depicts three models: Springfield, SpHealth, and SpCompanies. The first model imports the other two, creating a common name space. It shows the object homer with facets of types Employee and Patient, where the link surgeries is due to its facet of type Patient.

9.3.1. Comparison with other multi-view approaches. Next, we compare with some multi-view modelling approaches (see Table V). There are many proposals to handle multiple views

```
 1  FacetLaws for Census, Employment with MedicalRecords {
 2    must extend <p:Person & Employee>   // Persons that are employees get job insurance
 3      with healthRecord : Patient with {
 4        name = name [equality]
 5        jobInsurance : $ self.active implies self.insurance $
 6      }
 7    may extend <p:Person & !Employee>   // Persons that are not employees can get private insurance
 8      with healthRecord : Patient with {
 9        name = name [equality]
10        insurance = true
11        privateInsurance : $ self.insurance=true $
12      }
13    must extend <p:Person> where $ p.spouse.isDefined() and p.spouse.isKindOf(Employee) $
14      with healthRecord : Patient with {   // Spouses of employees get insurance from partners
15        name = name [equality]
16        partnerInsurance: $ self.spouse.insurance implies self.insurance $
```

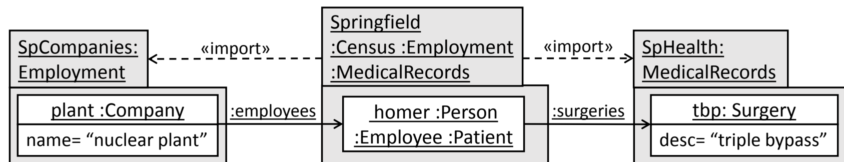


Fig. 18: Models depicting different views of Springfield.

(see [Bruneliere et al. 2019] for a recent survey), so we just revise some representative ones. We use the features identified in [Atkinson et al. 2015] as specific comparison criteria:

— *View types:* in *projective* approaches, there is a single underlying model (SUM) for the system from which views are derived using a projection/filtering mechanism. *Synthetic* approaches use no SUM, but the system is synthetized by integrating its views.
— *View correspondences:* views can be related by *explicit* traces, or their relation may be *implicit*.
— *Correspondence definition:* correspondences between elements of multiple views may be defined at the instance level (*extensional*) or at the type level (*intensional*).
— *SUM type:* in projective approaches, the SUM may be free of internal redundancies (*essential*), or it may be structured as an amalgamation of numerous, inter-related submodels which may store redundant information (*pragmatic*).

Table V: Comparison of approaches for multi-view modelling.

	Specific criteria				General criteria			
Approach	**View types**	**View corr.**	**Corr. definition**	**SUM type**	**R1**	**R2**	**R3**	**R4**
Central repository	synthetic	explicit	extensional	n/a	✓	×	×	×
OpenFlexo [Golra et al. 2016]	synthetic	explicit	extensional	n/a*	×	✓	∼	✓
OSM [Atkinson et al. 2010b]	projective	implicit	intensional	essential	×	✓	×	✓
Vitruvius [Kramer et al. 2013]	projective	implicit	intensional	pragmatic	✓	✓	×	✓
Facets	both	both	both	pragmatic	✓	✓	✓	✓

Our baseline for comparison is the central repository approach, which stores all models/views of a system in a central location. The views are synthetic, and correspondences must be explicitly managed at the level of models (extensional).

OpenFlexo [Golra et al. 2016] is a model federation framework that supports the gathering of data from base models expressed in heterogeneous technical spaces (EMF, XML, OWL, Excel, etc.) into virtual views. Correspondences are explicitly managed and specified at the level of models (extensional). Through views it is possible to propagate changes from one base model to the others, a feature that can be used to create the illusion of an essential SUM (hence the * in Table V).

Orthographic Software Modelling (OSM) [Atkinson et al. 2010b] implements the classic SUM approach, where views are dynamically generated based on transformations from and to the SUM. Hence, OSM is projective with implicit correspondences that are typically intensional. The SUM in OSM is minimalistic, hence it is an essential SUM.

In Vitruvius [Kramer et al. 2013], all the system information is accessed through views. It adheres to the ideas of OSM by providing means to construct and maintain a modular, virtual SUM consisting of individual models expressed in different modelling languages. Views are projections, and correspondences between elements are implicitly defined using one of three languages: an imperative language to specify uni-directional consistency enforcement from one view to another, a language for specifying bi-directional mappings, and a third language for specifying parameterized consistency invariants. The specifications are defined between view metamodels, and hence are implicit and intensional. The virtual SUM is pragmatic, as the views might contain duplicate information.

While facet-oriented modelling follows primarily a synthetic approach to view definition, our scenes also permit projecting selected facet information from objects. Facet laws can specify correspondences implicitly and intensionally (with the construct *must extend*), or allow a modeller to define them explicitly (using *may extend* and then the command **addFacet**). While it would be possible to build an essential SUM using facets, it is more likely that some facets duplicate information stored in other facets, hence we classify facets as providing a pragmatic SUM.

Regarding the general requirements of Section 2, the central repository approach, Vitruvius and facets are modular and non-intrusive (**R1**) as they permit connecting existing models externally. Instead, OSM and OpenFlexo are intrusive: OSM requires building a SUM metamodel, and OpenFlexo requires defining a virtual model that relies on role-based modelling [Drouot et al. 2019]. All approaches but the baseline enable objects to acquire types, fields and constraints (**R2**). In the central repository approach, this needs to be manually implemented. No approach but OpenFlexo and facets allows objects to change their type dynamically (**R3**). Instances in OpenFlexo can change their role on the fly, but this needs to be programmed at the implementation level [Drouot et al. 2019] (hence the $\sim$). Finally, all approaches except the baseline support some kind of bx field synchronization (**R4**).

9.4. Multi-level modelling

Multi-level modelling [Atkinson and Kühne 2001; González-Pérez and Henderson-Sellers 2006; de Lara et al. 2014; de Lara and Guerra 2018] enables the use of multiple metalevels instead of just two (models and metamodels) as is common in MDE practice. The use of multiple metalevels results in simpler models in scenarios that involve the type-object pattern or some of its variants [de Lara et al. 2014].

As an example, suppose we need a metamodel that allows the dynamic creation of product types (e.g., books) which have a value added tax (VAT), and products of those types which have a price. In addition, product types may define specific attributes (e.g., the number of pages), which receive a value in products.

Figure 19(a) shows a solution metamodel using standard modelling. It contains the classes ProductType and Product, a relation emulating a typing relation between them, and the classes Attribute and Slot to emulate the definition of attributes in ProductTypes

and their instantiation in Products. This way, the metamodel contains two instances of the *type-object* pattern [Martin et al. 1997]: ProductType/Product and Attribute/Slot. The

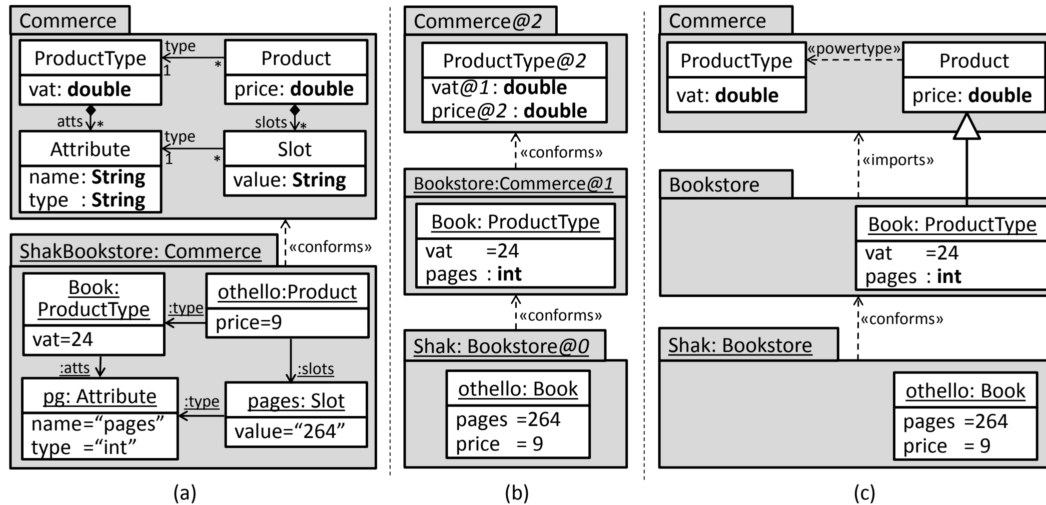


Fig. 19: Dynamic product types: (a) type-object pattern, (b) multi-level modelling, (c) powertypes.

Figure 19(b) shows an alternative multi-level solution. The top level declares the class ProductType, which is instantiated by Book one level below, which in turn is instantiated at the bottom level by othello. Since elements (like Book) can be types for the level below, and instances of the level above, they are called clabjects (by the contraction of the words class and object) [Atkinson 1997]. To characterize the instances beyond the next level, elements (models, clabjects, fields) can declare a *potency* [Atkinson 1997; Atkinson and Kühne 2001]. This is shown in the figure after the "@" symbol. The potency is a natural number, or zero, controlling how deep an element can be instantiated. Instances always have the potency of their type minus one. Elements with potency zero cannot be instantiated, and in case of attributes, they can receive a value. In the figure, model Commerce has potency 2, and so it can be instantiated in the two subsequent levels. Clabject ProductType declares vat with potency 1, so it receives a value at the next level, and price with potency 2, so it receives a value two levels below. Clabjects may declare new fields at any level. This is why Book can declare the new field pages. This is called linguistic extension in [de Lara et al. 2014].

Powertypes are another approach to multi-level modelling [Odell 1994; González-Pérez and Henderson-Sellers 2006]. A powertype is a type, whose instances are subtypes of another type. Figure 19(c) shows a solution to the example using powertypes. ProductType is the powertype of Product because the instances of ProductType (like Book) become subtypes of Product. This way, powertypes make explicit the dual type/object nature of elements by combining inheritance and instantiation.

The goal of this case study is to show that metamodelling frameworks enriched with facets can emulate multi-level modelling within two metalevels following a powertype approach. Figure 20 shows how to realize multi-level modelling using facets. We exploit the fact that facets can be added to classes and objects, and hence, we model the instance nature of elements by using facets, and their type nature by using classes. Moreover,

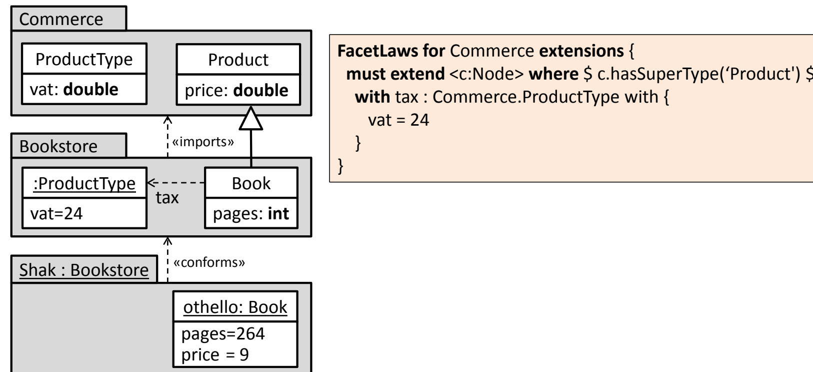


Fig. 20: Support for multi-level modelling using facets.

This example illustrates the use of facets at the metamodel level. In this case, the defined laws apply to extensions of the Commerce model (i.e., to models that import the Commerce model). Instead, when facets are used at the model level, the facet laws apply to instances of a certain metamodel (e.g., to instances of Census, as in Listing 26).

```
1  Model Commerce {
2      Node ProductType { vat : double; }
3      Node Product { price : double; }
4  }
5
6
7  Model Bookstore imports Commerce {
8      Node Book : Product {
9          pages : int;
10         vat = 24;
11     }
12 }
13
14
15 Bookstore Shack {
16    Book othello {
17        price = 9;
18        pages = 130;
19    }
20 }
```

Listing 27: Facet-based multi-level model.

```
1  Model Commerce@2 {
2      Node ProductType@2 {
3          vat@1: double;
4          price@2 : double;
5      }
6  }
7
8  Commerce Bookstore {
9      ProductType Book {
10         pages : int;
11         vat = 24;
12     }
13 }
14
15 Bookstore Shack {
16     Book othello {
17         price = 9;
18         pages = 130;
19     }
20 }
```

Listing 28: Potency-based multi-level model.

For comparison, Listings 27 and 28 show the facet-based and potency-based solutions to the example encoded in METADEPTH. Both solutions are similar, but with facets, the Commerce metamodel needs two classes to hold the attributes of ProductType and Product, while in the potency version, ProductType defines all attributes with a different potency. Model Bookstore is very similar in both solutions, but the facet version uses inheritance (written Book : Product, denoting that Book inherits from Product) instead of instantiation (written ProductType Book, denoting that Book is an instance of ProductType). The facet law adds a facet of type ProductType to Book, and so the assignment vat=24 in line 10 of Listing 27 is valid. Finally, model Shak is the same in both solutions.

Regarding field access, in the potency-based multi-level model, the expression othello.vat is valid because a lookup mechanism fetches fields first in the object, and if not

found, it traverses the type hierarchy, returning Book.vat in this case. Facets have a similar mechanism: if the facets of an object lack a field, this is fetched in the facets of the type. Finally, note that the use of facets is not restricted to models with three metalevels, but the powertype pattern can be iterated to consider arbitrary metalevels.

9.4.1. Comparison with multi-level modelling approaches. Table VI compares some multi-level modelling approaches – including the native potency-based support of METADEPTH – using the general requirements of Section 2 and specific criteria related to the handling of the classification relation [Atkinson and Kühne 2017; Jácome-Guerrero and de Lara 2020]:

— *Metalevels:* whether the approach supports splitting a model into an arbitrary number of metalevels or if such a number is fixed (e.g., 1 in case of EMF).
— *Potency:* if the approach supports potency as deep characterization mechanism [Atkinson 1997; Atkinson and Kühne 2001].
— *Multi-typing*: whether objects can have multiple types (including no type), or exactly one type is required.
— *Dynamic typing*: if the types of objects can change dynamically.

All approaches can use an arbitrary number of metalevels, but while the first three are potency-based, ML2 [Fonseca et al. 2018] and facets are based on powertypes. Altogether, facets can emulate approaches for multi-level modelling like powertypes. However, the facet-based mechanism is more powerful than both potency and powertypes in two aspects: facets can be added and deleted dynamically, and elements can have several facets. This means that the type of an element can change dynamically depending on conditions established in facet laws, and that an element can have several types. In contrast, inheritance and typing are static in standard potency-based multi-level modelling and powertypes, and objects have at most one type. As an exception, MultEcore supports supplementary typing dimensions that cannot change dynamically.

Table VI: Comparison of approaches for multi-level modelling.

	Specific criteria				General criteria			
Approach	**Metalevels**	**Potency**	**Multi-typing**	**Dynamic typing**	**R1**	**R2**	**R3**	**R4**
METADEPTH (without facets)	arbitrary	yes	no	no	✓	✓	∼	×
Melanee [Atkinson and Gerbig 2016]	arbitrary	yes	no	no	✓	×	×	×
MultEcore [Macías et al. 2018]	arbitrary	yes	yes	no	✓	✓	×	×
ML2 [Fonseca et al. 2018]	arbitrary	no	yes	no	✓	✓	×	×
Facets	arbitrary	no	yes	yes	✓	✓	✓	✓

Regarding the general requirements, all approaches to multi-level modelling are modular (**R1**). Acquiring new types, fields and constraints (**R2**) is possible in METADEPTH as it supports a-posteriori typing; in MultEcore by means of supplementary typing dimensions; and in ML2 where it can be emulated with inheritance. Only facets and partially METADEPTH support dynamicity (**R3**). Finally, the maintenance of field relations is only available in facets (**R4**).

9.5. Language product lines

This section demonstrates the use of facets to define language families, based on the notion of product line (PL). A PL is a set of related software artefacts, each implementing some variation [Northrop and Clements 2002]. The variability space is often defined through a feature model [Kang et al. 1990] which specifies the available features and their allowed combinations. Then, different techniques permit deriving an artefact out of the chosen feature configuration.

PLs have been used in MDE to describe variants of modelling languages [White et al. 2009; Méndez-Acuña et al. 2016; Guerra et al. 2020], model variants [Czarnecki and Pietroszek 2006] and transformation variants [de Lara et al. 2018a]. Here, we focus on PLs of modelling languages (more specifically, PLs of metamodels) whose configuration can change at run-time and trigger a model reconfiguration. Such dynamic PLs are useful to realize models at run-time for adaptive applications [Blair et al. 2009], where a model of the running (adaptive) system is used at run-time with the purpose of analysis, reasoning or monitoring.

To illustrate the use of facets to realize this scenario, we are defining a simple PL of a component-based language where components have input and output ports, and the latter ports can be connected to the former. Listing 29 shows the base metamodel for this language PL.

```
1  Model Components {
2    abstract Node NamedElement {
3      name : String;
4    }
5
6    Node Component : NamedElement {
7      ports : Port[1..∗];
8    }
9
10   abstract Node Port : NamedElement ;
11
12   Node InputPort : Port;
13
14   Node OutputPort : Port {
15     target : InputPort[1..∗];
16   }
17 }
```

Listing 29: Components metamodel.

```
1  Model ComponentFeatures { // feature model
2    Node FeaturedElement {
3      security : boolean = false;
4      monitoring : boolean = true;
5    }
6  }
7
8  Model ComponentFacets { // facet metamodel
9    Node Cipher {
10     blockSize : int;
11     key : String;
12     nRounds : int;
13   }
14   Node Monitor {
15     activeRate : double = 0.0;
16     powerConsumption : double = 0.0;
17   }
18 }
```

Listing 30: Feature model and facets for the variants.

There are many variants of component-based languages [Malavolta et al. 2010]. For simplicity, we just consider two features in the PL, *security* and *monitoring*, which can be combined to yield four language configurations. Lines 1–6 of Listing 30 define the features in a class named FeaturedElement. If needed, the class can include constraints to forbid incompatible feature combinations. Next, lines 8–18 of Listing 30 declare a metamodel with the facets that will be compositionally added to component models upon the selection of features. Choosing the feature *security* implies providing the ports with parameters needed to cipher the messages (we assume the RC5 algorithm, which requires defining a block size, a key size and a number of rounds [Rivest 1994]). Choosing *monitoring* adds a (shared) monitor to the system to calculate run-time statistics. Our approach is compositional and uses positive variability: when a configuration is selected, facets of types Cipher and Monitor can get woven into component models.

We use the facet law specification in Listing 31 to automate the management of facets. Its first law (lines 1–6) creates a facet of type FeaturedElement (defined in Listing 30) to hold the selected feature configuration. This facet is shared by all elements of the component model. The second law (lines 8–20) adds a Cipher facet to Ports when the feature *security* is selected, and a Monitor shared facet to Components when the feature *monitoring* is selected. While the law conditions check for the value of a feature, any arbitrary logic formula can be required.

The adaptations occur at the model level, and the metamodels for the components and the facets are never explicitly woven together, remaining modular. This PL is dynamic because, when the feature configuration of a model is changed by setting the fields *security* or *monitoring* to a different value, appropriate facets are added to or removed from all model objects accordingly.

```
1  FacetLaws for Components with ComponentFeatures {
2      must extend <n:NamedElement> with cfg: FeaturedElement with {
3          security = false
4          monitoring = true
5      } reuse
6  }
7
8  FacetLaws for Components, ComponentFeatures with ComponentFacets {
9      must extend <p:Port & FeaturedElement> where $ p.security $
10         with c : Cipher with {
11             blockSize = 32
12             key = "915F4619BE41B2516355A50110A9CE91"
13             nRounds = 12
14         }
15     must extend <c:Component & FeaturedElement> where $ c.monitoring $
16         with m: Monitor with {
17             activeRate = 0
18             powerConsumption = 0
19         } reuse
20 }
```

Listing 31: Facet law specification for component PLs.

9.5.1. Comparison with other model-based and language PL approaches. Table VII compares several approaches to define language variability, focusing on the general requirements **R1**–**R4** and the following specific aspects:

— *Level* at which the adaptation occurs. This may be at the *meta* level, to obtain variants of a language description; or at the *model* level, to obtain variants of a model without changing the language description.
— *Variability* refers to the mechanism to express the adaptation. It is *negative* if the initial artefact contains all variants and the unselected ones are removed; *positive* if the variants are separate and the selected ones are woven together; and *transformational* if it supports defining creation and deletion operations over a base artefact.
— *Adaptation* of the artefact can be either *explicit* (e.g., through weaving) or *implicit*.
— *Dynamicity* indicates whether run-time reconfigurations of a selected variant are supported (where the working models are adapted to the new language features), in addition to adaptations at design time (where models are not adapted).

Table VII: Some model-based and language PL approaches.

	Specific criteria				**General criteria**			
Approach	**Level**	**Variability**	**Adaptation**	**Dyn.**	**R1**	**R2**	**R3**	**R4**
Model templates [Czarnecki and Pietroszek 2006]	meta, model	negative	explicit (PCs)	no	×	✓	×	×
DeltaEcore [Seidl et al. 2014]	meta, model	transformational	explicit (DSL)	no	✓	✓	×	×
VML* [Zschaler et al. 2009]	meta, model	transformational	explicit (DSL)	no	✓	✓	×	×
SmartAdapters [Perrouin et al. 2012]	meta, model	negative	explicit (aspects)	no	×	✓	×	×
Metamodel PLs [Guerra et al. 2020]	meta	negative	explicit (PCs)	no	×	✓	×	×
Featured model types [Perrouin et al. 2016]	meta	negative	explicit (PCs)	no	×	✓	×	×
Facets	meta, model	positive	implicit (meta) explicit (model)	yes	✓	✓	✓	✓

Model templates [Czarnecki and Pietroszek 2006] can be applied to metamodels or models, but adaptations do not affect lower levels, so the existing instances are not co-evolved. The variability space is specified through a feature model. This variability is negative: there is a so-called 150% model where all variants are superimposed, and its elements can define boolean formulae over the features, called presence conditions (PCs). When a feature configuration is selected, the elements whose PC evaluates to

false are deleted from the 150% model to obtain a variant. The approach is not dynamic, since model variants cannot be reconfigured once generated.

DeltaEcore [Seidl et al. 2014] is an approach to generate delta languages for specific metamodels. A delta language permits specifying transformations to be executed over models when a feature configuration is selected. The approach is similar to VML* [Zschaler et al. 2009]. None of them are dynamic.

The SmartAdapters [Perrouin et al. 2012] approach weaves metaclasses into the domain metamodel to obtain variability on its instances. As in the previous cases, by weaving the variability elements at the meta-metamodel, it is possible to obtain a metamodelling language with possibilities to define variability.

Metamodel PLs [Guerra et al. 2020] and feature model types [Perrouin et al. 2016] target the definition of PLs of metamodels. In both cases, the variability is negative, and the elements of the metamodel that contains all possible variants can define PCs. None of them consider the adaptation of existing models when the metamodel is reconfigured, and choosing a new configuration requires re-generating the metamodel variant.

In contrast to the previous approaches, when using facets to model language variability, the language definition does not change (i.e., the creation and facet metamodels are not woven together) but the models do change (through the acquisition and loss of facets). This approach is dynamic because whenever a new configuration is chosen, the model is reconfigured. These characteristics make facets a suitable basis for model-based dynamic PLs [Capilla et al. 2014; Cetina et al. 2009].

Regarding the general requirements, all approaches based on negative variability are intrusive (**R1**) as they require either adding PCs to existing (meta)models (metamodel PLs, featured model types) or weaving variability primitives on the metamodels (SmartAdapters). Transformational approaches are non-intrusive since products are derived using external languages. All approaches permit objects to take new types, fields and constraints (**R2**), but only facets provide dynamicity (**R3**). While all approaches permit assigning values to object fields, only facets support the maintenance of field relations (**R4**).

9.6. Discussion

To conclude, we reflect on the benefits and limitation of facets based on the case studies.

9.6.1. Benefits of facet-oriented modelling, and answer to the research question. So far, we have explored the use of facets on a running example and five scenarios. Table VIII summarizes the benefits that using facets brings to them.

Table VIII: Benefits of facets in relation to the case studies.

Case study	Facilitated activity	Benefits
Information integration (running example)	Model/metamodel extension	Extended (meta)models do not change, seamless access to extensions, dynamicity, field synchronization
Annotation models	Model extension	Declarative specification of annotations, seamless access to annotations, dynamicity, annotation-field synchronization
Transformation reuse	Model adaptation	Adaptation is virtual (does not create a separate adapted model), in-place, bx
Multi-view modelling	Viewpoint/view definition View consistency management	Declarative specification of consistency relations between types and fields
Multi-level modelling	Add instance features to types	Multiple typing, dynamic typing, dynamic acquisition/loss of instance features
Language product lines	Define language variants	Dynamic selection of variants, automatic model adaptation

In the running example, an existing Census model is enriched with Employment information. This entails both metamodel extension (facet metamodels "virtually" extend the creation metamodel) and model extension (facets are acquired and dropped by host objects). Our solution based on facets does not require modifying existing metamodels

or creating new ones, the facet types and fields can be seamlessly accessed from the host objects, model extension is dynamic, and fields can be synchronized via adapters.

The case study on annotation models (Section 9.1) deals with the extension of models with additional data. When using facets, these data can be added and removed dynamically to the model objects based on declarative conditions. Moreover, the access to annotations is transparent, and there is the option to synchronize annotations and host object fields via adapters.

For transformation reuse (Section 9.2), facets allow adapting models "virtually" so that they can be seen as instances of the metamodel the transformation works on. The advantage is that this adaptation is in-place and bx, i.e., there is no need to build a separate adapted model and then translate changes in the adapted model back to the original one. As we will see in Section 9.6.2, this improves the performance.

Multi-view modelling (Section 9.3) involves the definition of viewpoints, views and consistency relations. Facet-oriented modelling can be seen as an alternative to other existing approaches, by which facet metamodels define viewpoints, scenes produce views, and facet laws stablish consistency relations via constraints and reactive field adapters that maintain field values synchronized.

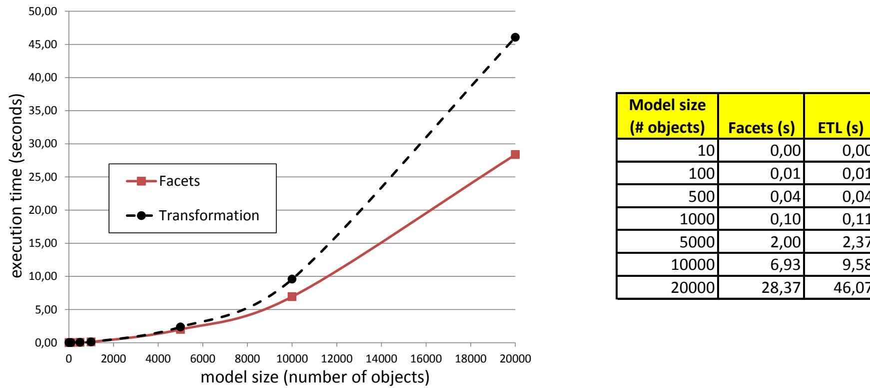
In multi-level modelling (Section 9.4), clabjects are both types and instances. Facets enable multi-level modelling within two-level architectures, as classes can be added facets with slots, so that they effectively become clabjects. Moreover, clabjects can have several facets (multiple typing) dynamically added and removed (dynamic typing).

As for language product lines (Section 9.5), they require defining and selecting variants of a language. The advantage of using facets for this purpose is being able to select the variants dynamically, so that the models are automatically adapted without having to explicitly weave metamodel variants or slice a base metamodel.

Altogether, we have compared facets against 25 approaches to solve 6 different scenarios. Most of the approaches have been built to solve a specific scenario, like transformation reuse. A few approaches, like a-posteriori typing, can be applied to several scenarios (information integration, model annotation, transformation reuse), but cannot solve them all (multi-view modelling, multi-level modelling, language product lines). In contrast, a facet-based approach is able to handle all of them without requiring ad-hoc extensions. This shows that adding a dynamic object adaptation mechanism atop modelling frameworks avoids the need to construct dedicated tools to support these diverse scenarios. Hence, we can answer positively the research question formulated in the introduction (*can a dynamic mechanism for object adaptation be used to solve MDE scenarios that are currently tackled using dedicated, ad-hoc tooling?*).

9.6.2. Quantitative evaluation. Next, we analyse whether a facet-based solution to a given scenario can be as performant as other techniques. Even though our current implementation is not optimized for performance, this evaluation gives a rough estimation and can serve as a reference for tool builders wanting to port facets to other modelling frameworks, like EMF. Our goal is not to be exhaustive, but rather to identify possible performance issues.

First, we compare the cost of model adaptation either by adding facets to the model or by creating a separate adapted model (Section 9.2). For this purpose, we adapted models of increasing size (from 10 to 20000 objects) using two approaches: applying the facet law of Listing 23, and creating an adapted model via an ETL transformation executed within METADEPTH. The models contained 40% objects matching the facet law/transformation. We used a Windows 7 computer with an intel i7-2600 CPU and 12GB of RAM, and run each experiment 7 times, taking the median. The results, displayed in Figure 21, show that both techniques perform similarly for small models, while for big models, the overhead of creating a new model is higher than using facets.



Model size (# objects)	Facets (s)	ETL (s)
10	0,00	0,00
100	0,01	0,01
500	0,04	0,04
1000	0,10	0,11
5000	2,00	2,37
10000	6,93	9,58
20000	28,37	46,07

Fig. 21: Execution time for facet laws vs. an ETL transformation, both running on METADEPTH.

The access to facet fields relies on delegation (see Figure 15). This incurs a higher cost than accessing a field owned by the host object. We have quantified this penalty by monitoring an EOL program that accesses 1000 times a facet field on a host object, and an object field. We repeated this experiment 70 times, taking the median. Overall, accessing an object field required 0,028 ms, and the time to access a facet field was 0,04 ms (an increase of 0,012 ms). This time increases when the delegation chain has several navigation steps (i.e., facets with facets). While this penalty time may be acceptable, it can be reduced by caching the facet fields in the host object. However, this would increase the facet creation time.
Threats to validity. We have used just one facet law to measure the times in Figure 21, so other law specifications may yield different results. However, the law characterizes typical facet behaviours (creating a facet, reusing a facet).

9.6.3. Limitations of facet-oriented modelling. As we have seen in the previous section, accessing facet fields from a host object incurs in a penalty time, although this overhead can be reduced by caching the facet fields in the object. Moreover, given a program, it is difficult to statically type check that any defined access to facet types and fields is legal, as these are acquired dynamically. Instead, dynamic type checking (e.g., in the style of [Millan et al. 2017] for OCL) is required. This is consistent with the dynamic nature of prominent model management languages, like ATL [Jouault et al. 2008] and the Epsilon languages [Paige et al. 2009].

Regarding the analysis methods presented in Section 7, they are semi-decidable as they perform a bounded search and hence may miss solutions that fall outside these bounds. However, these analyses rely on the “small scope hypothesis” assumption [Jackson and Damon 1996; Andoni et al. 2003] which argues that most solutions can be found within some small scope. Another limitation of the analysis of reactive adapters is that it currently works just for fields whose value may change when traversing twice a cycle of dependencies. This captures some problems like non-converging field values or fields which take alternating values. However, networks of transformations may have other problems [Klare et al. 2019]. We leave to future work the improvement of our analyses with more sophisticated techniques.

Regarding practicality, we would like to investigate how software engineers react to facet-oriented modelling by conducting a user study. Our plans also include testing our approach on larger case studies, and analysing its application to other MDE artefacts like code generators.

10. RELATED WORK

In this section, we discuss additional related works to compare facets with the notion of role in conceptual modelling (Section 10.1); with prototype, delegation-based and reactive languages (Section 10.2); and with specific MDE techniques (Section 10.3).

10.1. Roles

Facets are similar to roles and fulfil most features of role languages [Kühn et al. 2015; Steimann 2000; Kühn et al. 2014; Jodlowski et al. 2003; 2004]. Table IX summarizes these features. Features #1–#14 were identified by Steimann [Steimann 2000], and #15–#18 have been added by us to illustrate the differences between roles and facets.

Table IX: Comparison of some modelling approaches supporting roles. The symbol '∼' means that either the feature is not native but reasonably possible, or it is partially supported. N.A. means not applicable.

#	Feature	Facets	Lodwick	CROM	ORM 2	Entity Types	Roles on VODAK
1	Roles have properties and behaviours	y	y	y	n	y	y
2	Roles depend on relationships	y	y	y	y	y	y
3	Objects may play different roles simultaneously	y	y	y	y	y	y
4	Objects may play the same role several times	y	y	y	y	n	y
5	Objects may acquire and abandon roles dynamically	y	y	N.A.	y	y	y
6	The sequence of role acquisition/drop may be restricted	y	∼	y	∼	y	y
7	Unrelated objects can play the same role	y	y	y	n	n	n
8	Roles can play roles	y	n	n	n	y	y
9	Roles can be transferred between objects	∼	∼	N.A.	∼	∼	∼
10	The state/features of an object can be role-specific	y	∼	y	n	n	y
11	Roles restrict access	y	N.A.	N.A.	∼	y	y
12	Different roles may share structure and behaviour	y	∼	n	n	y	y
13	An object and its roles share identity	y	y	y	y	n	n
14	An object and its roles have different identities	y	n	y	n	y	y
15	Facets/roles bring types to objects (**R2**)	y	y	n	n	n	n
16	Facets/roles can be applied at any metalevel	y	n	n	n	n	n
17	Facets/roles are modular and non-intrusive (**R1**)	y	n	n	n	n	n
18	Automated maintenance of relations between owned and acquired fields (**R4**)	y	n	n	n	n	n

While most features are typical of all role approaches, some are more specific. Steimann [Steimann 2000] describes them as follows. Roles have properties and can be assigned behaviours (#1). Roles are relevant in the context of a relation with an object (#2). Objects may play different roles at the same time (#3) and the same role multiple times (#4), e.g., a Person with several employments. Roles are dynamic, so that objects may acquire and drop roles during their life-cycle (#5). Developers may need to specify valid sequences of role acquisition/drop (#6). Objects of unrelated types can play the same role (#7). Roles can play roles (#8), e.g., an employee – the role of a Person – can play the role of a project manager. In some approaches, a role dropped by an object can be acquired by another (#9). The state and features of an object can be role-specific (#10), which means that the object acquires the properties of the role. When an object is accessed using the role, some of its features remain invisible (#11). For example, from an Employee role of Person, one cannot access the Person age. Roles can be defined using inheritance-like mechanisms (#12). Roles can be part of an object and share identity (#13), or alternatively, an object and its roles have different identity (#14).

Features #15, #17 and #18 come from requirements **R1–R4** in Section 2, and feature #16 shows a difference between facets and some role approaches. Feature #15 (part of **R2**) states that facets bring types to their host objects, which entails multiple classification. While according to feature #3, objects can play several roles at a time, some approaches like *EntityTypes* [Cabot and Raventós 2006] do not support multiple classification. **R2** also requires facets/roles to bring properties and behaviour to objects,

but this is covered by feature #1. Facets can be applied at any metalevel (#16), but in contrast, most role approaches have been devised for the model level. We can use the *creation* and *facet* metamodels in a modular, non-intrusive way (#17, **R1**) as facet laws can specify relations without modifying these metamodels. Instead, as roles depend on relationships (#2), some role approaches need to modify the metamodels to add those relationships. Finally, feature #18 (**R4**) is the ability to establish and automatically maintain relations between fields of facets and objects.

In the following, we focus on the most representative role-based modelling approaches (see [Cabot and Raventós 2006] for a comparison with further works). *Lodwick* [Steimann 2000] is a theoretical proposal that unifies features of several role-based languages. The theory is based on declaring inheritance hierarchies for natural and role types defined as distinct relations, and on an explicit *role-filler* relation indicating which natural types can play which role type. Being a theory, it does not have a practical implementation, and does not consider model management operations as we do. One salient feature of *Lodwick* is that roles are not instantiated, but their properties are taken by the instances of the natural types, hence roles and objects share identity (features #13 and #14). Regarding features #15–#18, roles bring their type to the object. They are defined at the class-level and instantiated one level below, hence they cannot be applied at arbitrary metalevels. *Lodwick*'s formalization distinguishes between natural and role types and requires explicit relations, so existing metamodels cannot be used modularly. Finally, there is no equivalent to our field adapters.

ORM 2 [Halpin 2005] is a well-established modelling language, but it sees roles as relationship ends, similarly to languages such as UML or MOF. It does not fulfil any of the specific features of facets (#15–#18).

Dahchour et al. propose a role model, and a metaclass-based implementation on top of the VODAK modelling language [Dahchour et al. 2004]. The approach is based on establishing a special role relationship between object and role classes. As this relation is binary, a role class is related to exactly one object class (feature #7). Role classes are instantiated to become part of objects, which then can access their properties through delegation. Role acquisition and loss can be controlled by specifying meaningful role combinations and transition predicates. Being based on single classification, roles cannot bring their types to the objects (#15), roles cannot be applied at the class-level (#16), role definitions cannot be used modularly since declaring explicit relationships is required (#17), and there is support for field adapters (#18).

Cabot et al. define several conceptual modelling patterns to emulate roles [Cabot and Raventós 2006], like using names in plain associations, subtypes, interfaces and association classes. They present a solution called *EntityTypes* based on stereotypes, by which associations can be stereotyped as «roleOf» and declare the properties the role adopts from the class. In addition, stereotyped operations can establish the conditions for role acquisition and drop. The authors do not provide automated support, but describe how to implement the patterns in Java. Regarding features #15–#18, since Java does not support multiple classification, roles do not bring types to objects. Moreover, objects cannot access the role properties transparently, roles are not metalevel independent, they cannot be used modularly but require declaring explicit metamodel relations and operations, and there is no mechanism to synchronise fields.

CROM [Kühn et al. 2015; Kühn et al. 2014; Kühn et al. 2016] is a more recent proposal to cover most features of role modelling languages. It provides a graphical editor for role types [Kühn et al. 2016], but the support for role instances is still incipient. While it is a rich language with sophisticated concepts (compartment types, constraint groups), it lacks some essential elements for its use in MDE, like inheritance, explicit attribute handling, OCL constraints, or integration with model management languages as facets provide. Regarding features #15–#18, roles do not bring types to the object, cannot

be applied at any metalevel, cannot be used modularly (since explicit role types and relations need to be defined), and no synchronization of fields is possible.

Regarding facets, the main limitation is the lack of native support for transferring facets between objects, but this would be easy to add. While features #13 and #14 are normally consequences of different design decisions, we support idioms to model both. On the one hand, facets can be seen as parts of host objects and therefore share identity with them. On the other hand, granulated scenes promote facets to objects, hence providing distinct identities.

10.2. Prototype, delegation-based and reactive programming languages

Some programming languages support incremental class definitions, like open classes (e.g., Ruby) and partial classes (e.g., C#). In our case, facets can work both at the class and object levels. Several programming languages have experimented with roles. For example, in [Pradel and Odersky 2008], roles are emulated in Scala using a lightweight mechanism based on compound objects with dynamic proxies, which avoids changing the language. This increases flexibility by enabling objects to change their type. Regarding facet acquisition, our laws permit specifying declarative conditions for acquiring facets, while programming languages use imperative commands (which we also support with **addFacet/removeFacet**). Moreover, we handle constraints, field relations, and can configure the usage of facet metamodels by means of interfaces.

Objects in dynamic programming languages are open: it is possible to add and remove fields at run-time. The advantages are reduced code size and increased reuse and flexibility [Cox et al. 2014] at the cost of less static assumptions and difficulty of analysis. Some modelling systems, like METADEPTH, permit adding fields to objects at design time [de Lara and Guerra 2010]. Instead of dealing with single fields, facets are regular objects that encapsulate fields, constraints and typing. Moreover, as facets are defined by a metamodel, defining operations over them is simple.

Prototype and delegation-based programming languages share similarities with facets [Lieberman 1986; Dony et al. 1992]. Different from class-based languages, objects are created "ex-nihilo" (from scratch) by cloning a prototypical instance or by extending an existing object. The languages often rely on delegation, so that if an object does not understand a method, it delegates the method call to its parent. Examples of delegation-based languages are SELF [Ungar and Smith 1987] and JavaScript. Facets can also be seen as a prototype-based mechanism with implicit delegation. However, facets have characteristics not found in this kind of languages, such as the ability to provide a type (leading to multiple classification), create facets based on classes, define conditions for facet acquisition, and automate field synchronization.

Some approaches combine classes and prototypes. For example, in [Bettini et al. 2011], incomplete classes can be instantiated to yield incomplete objects. These can be composed with complete objects to yield new complete objects at run-time. Method body lookup uses similar mechanisms to delegation. Object specialization [Sciore 1989] allows creating both class and object inheritance hierarchies. Hierarchies at the object level can be dynamically changed, and objects can have multiple parents. This approach is similar to ours, but we also propose facet laws, interfaces, and field adapters. Furthermore, facets can be applied to any metalevel, and can emulate clabjects (cf. Section 9.4).

Regarding reactive languages [Bainomugisha et al. 2013], they have primitives to express computations as reactions to external events. Examples of reactive systems include user interfaces (which react to user inputs) and embedded systems (which react to hardware signals) [Salvaneschi et al. 2015]. A prototypical example of reactive language is the spreadsheet (where cells are updated upon changes in other cells) but many libraries and languages have reactive capabilities [Bainomugisha et al. 2013; Salvaneschi et al. 2014; Elliott and Hudak 1997]. Different from event-based

languages, reactive languages support the declarative specification of reactive values, their composition, and automatic update [Salvaneschi et al. 2014]. Reactive languages may follow a *push* or a *pull* model of reaction [Bainomugisha et al. 2013]. The former is eager (reactions occur as soon as possible) and the latter is lazy (there may be latency between event and reaction). Some reactive languages support multidirectionality, so given an expression like $F = C*2+7$, whenever C or F change, the other one is updated. Reactive languages may require lifting operators (like + or *) or functions to operate on continuous time-varying values. Facets also support the definition of reactive field adapters to specify field value recomputations in reaction to changes. We implement a push model, as field changes immediately trigger the evaluation of the appropriate field adapters. The syntax of our adapters is not multidirectional, but there must be an adapter for each variable that needs recomputation (e.g., we would need to specify $F = C * 2 + 7$ and $C = (F - 7)/2$ in the previous example). Finally, lifting an expression to a reactive one is done by enclosing the expression between "[" and "]".

10.3. MDE techniques

Next, we compare facets with works specifically devised for MDE. We revise techniques for metamodel integration or extension, approaches on model extension or merging, and related works on (bx) model transformation languages and analysis techniques.

10.3.1. Metamodel-level techniques. Several researchers have proposed techniques to extend metamodels and sometimes co-evolve their instances.

In [Burmester et al. 2004], the authors propose patterns for metamodel extension and metamodel integration that facilitate (modelling) tool interoperability. The patterns exploit the possibility to extend a base metamodel via inheritance. The approach is implemented in Fujaba, which has a core metamodel that needs to be extended to create domain-specific metamodels. The approach is static (extensions cannot be added and dropped dynamically), object field access is not transparent (one needs to navigate bx associations), and multiple types are not allowed.

The lightweight metamodel extension mechanism proposed in [Brunelière et al. 2015a] targets metamodel evolution. It provides a list of metamodel extension operators (addition, update, filtering) and a DSL for specifying the extensions. The extensions keep existing models compatible and avoid migrations, since models are extended virtually. The approach requires the definition of a new concept – the metamodel extension – and does not allow objects with several types or dynamic types (extensions are not based on conditions). While facets can be shared by several host objects, and a host object can have several facets of the same type, this is not possible with this approach.

EMF-facet [Madiot and Dupe 2018; Pepin et al. 2018] is an Eclipse project to extend metamodels externally. Extensions are defined for individual classes using so-called EFacets that can add read-only derived fields, and optionally, a subtype to a class. The goal is producing customized tabular views of models. Instead of introducing new concepts (the EFacet), our facets are created using regular metamodels. Facets can be acquired or dropped both manually or automatically via facet laws, and provide a type, full-fledged fields and constraints to the host objects. Facets are objects and can be shared, which is not possible with EFacets. Finally, our facets are transparent to the model management languages, while this is not the case for EMF-facet, which persists extended models using a specific tabular metamodel.

As discussed in Section 9.1, EMF profiles [Langer et al. 2012] adapt the notion of stereotype to metamodels. This permits applying profiles to metamodels and extending models using the profile information. However, their working scheme is based on models with cross-references, with no support for transparent field access, dynamicity or field synchronization (cf. Table II).

Wende et al. take inspiration from roles to enable modular language definition [Wende et al. 2010], including the abstract syntax, concrete syntax and semantics. This way, metamodels defining the abstract syntax can declare role classes as placeholders to enable composition with other metamodel classes. Such role classes may declare operations that the bound metamodel classes need to implement. Hence, roles act like interfaces which are bound to classes and then statically woven.

10.3.2. Model-level techniques. A second set of approaches work with standard metamodels, and propose techniques to define extensions or views of models.

Barbero et al. [Barbero et al. 2007] propose the relation "extensionOf" between models, so that the extension model cross-references the base model. The extension model conforms to a metamodel that extends the base one, and a composition operator merges them. In a similar vein, decorator models [Kolovos et al. 2010] decouple a model from its annotations, which are persisted separately, and a delegation mechanism provides transparent access to the decoration fields (cf. Section 9.1). Both approaches are static (extensions/decorators at the model level cannot be obtained or dropped dynamically) and do not support multiple types or automatic field synchronization.

Model merging languages, like EML [Kolovos et al. 2006b], permit comparing two models of the same or different metamodels, and produce a new model. While model merging has similarities with facet-based modelling, the crucial point is that a facet and its hosts can be instances of different metamodels, and the hosts acquire the facet fields. This is not possible in standard model merging, as the extended object would not conform to the metamodel. Moreover, facets are dynamic.

EMF views [Brunelière et al. 2015b] offers a dedicated language and tool for defining views on potentially heterogeneous models. The approach adapts the concept of database views to models. View models behave as any regular model, but views are not materialized. A DSL similar to SQL enables defining viewpoints and generating views. Compared to our approach, EMF views are static and do not support multiple types or automatic field synchronization via adapters.

10.3.3. Model transformation and analysis. The MDE community has proposed a plethora of approaches, languages and tools to manipulate models [Kahani et al. 2019; Hidaka et al. 2016; Czarnecki and Helsen 2006]. Our language to manipulate facets can be considered a domain-specific transformation language [Sánchez Cuadrado et al. 2014]. The **addFacet** and **removeFacet** commands are rule-based, with selection patterns based on object identifiers, OCL queries, or tuples. Laws are a higher level specification mechanism, enforced via **addFacet** and **removeFacet** commands. This is reminiscent of the OMG standard QVT-Relational (QVT-R) transformation language [QVT 1.3. 2016], where specifications can be used to check the consistency of a set of models, and be transformed into QVT-Core for execution. In addition, facets can be shared between objects. Some transformation languages also have the ability to reuse objects created by different rules, more prominently QVT-R and its *check-before-enforce* policy.

Our reactive field adapters are used to maintain field values consistent. This feature is typical of bx transformation languages. For example, some triple graph grammar approaches use bx constraints to specify field relations [Anjorin et al. 2012]. Instead, our approach requires specifying a reactive adapter per field, and the support for bx adapters is future work. Reactivity has also been investigated in the context of model transformations, specifically for ATL [Perez et al. 2017]. That works aims to perform minimal recomputations in a target model when the source model changes. Instead, the goal of reactive field adapters is to keep fields in host objects and facets synchronized.

Our analysis techniques rely on model finding and the construction of an analysis metamodel. This is similar to the notion of *transformation model* [Bézivin et al. 2006], which has been used in the MDE community to analyse model transformations [Cabot

et al. 2010]. In our case, we need to deal with the semantics of facet reuse, and analyse well-behavedness of reactive adapters. This latter analysis permits detecting potential non-terminating cycles of updates. However, researchers have investigated and classified other error types that may arise in networks of transformations [Klare et al. 2019]. We plan to provide more comprehensive analysis support for reactive adapter specifications in the future.

11. CONCLUSIONS AND FUTURE WORK

In this paper, we have introduced facets as a way to add flexibility to modelling. In contrast to standard MDE practice, facets effectively make objects *open*, so that they can acquire and drop types, fields and constraints. We provide management commands for the opportunistic reuse of metamodels to create facets; facet interfaces and laws for planned facet-oriented modelling; and analysis mechanisms to check the applicability and satisfiability of facet laws, and to discover ill-behaved field adapters. We have discussed scenarios where facets have advantages with respect to current solutions, and shown an implementation atop METADEPTH.

Some elements of our approach can be improved. At the technical level, we would like to extend the triggered constraints mentioned in Section 8 so that they do not need to be type-local. This will require a static analysis of the facet law conditions. At the conceptual level, our field adapters are unidirectional, and so, two adapters are required to synchronize two fields (except for equality). Inspired by lenses [Foster et al. 2007], we aim at developing mechanisms that enable bidirectional synchronization expressions. Our analysis techniques can also be strengthened, e.g., checking conflicting constraints in interfaces.

Facet-oriented modelling brings dynamicity to modelling and opens the door to new possibilities. Our next goal is a deeper exploration of facets to solve other MDE problems, like hierarchical modelling. In this paper, we have been concerned mostly about model structure, and so the next focus will be on behaviour. We are working on suitable concepts – akin to the notions of method redefinition and delegation in object orientation – for combining operations defined in the host and facet objects. We also plan to develop static analysis techniques for the analysis of model management programs that manipulate models, especially in-place transformations.

Acknowledgements. Work partially funded by the R&D programme of the Madrid Region (project FORTE, S2018/TCS-4314), and the Spanish Ministry of Science (project MASSIVE, RTI2018-095255-B-I00). We thank the anonymous referees for their useful comments, which helped us improving the paper.

REFERENCES

Alexandr Andoni, Dumitru Daniliuc, and Sarfraz Khurshid. 2003. *Evaluating the "small scope hypothesis"*. Technical Report MIT-LCS-TR-921. MIT CSAIL.

Anthony Anjorin, Gergely Varró, and Andy Schürr. 2012. Complex attribute manipulation in TGGs with constraint-based programming techniques. *ECEASST* 49 (2012).

Colin Atkinson. 1997. Meta-modeling for distributed object environments. In *EDOC*. IEEE Computer Society, 90–101.

Colin Atkinson and Ralph Gerbig. 2016. Flexible deep modeling with Melanee. In *Modellierung (LNI)*, Vol. 255. GI, 117–122.

Colin Atkinson, Bastian Kennel, and Björn Goß. 2010a. The level-agnostic modeling language. In *SLE (LNCS)*, Vol. 6563. Springer, 266–275.

Colin Atkinson and Thomas Kühne. 2001. The essence of multilevel metamodeling. In *UML (LNCS)*, Vol. 2185. Springer, 19–33.

Colin Atkinson and Thomas Kühne. 2017. On Evaluating Multi-level Modeling. In *Proceedings of MODELS 2017 Satellite Event (CEUR Workshop Proceedings)*. CEUR-WS.org, 274–277.

Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2010b. Orthographic software modeling: A practical approach to view-based development. In *Evaluation of Novel Approaches to Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 206–219.

Colin Atkinson, Christian Tunjic, and Torben Moller. 2015. Fundamental realization strategies for multi-view specification environments. In *EDOC*. IEEE Computer Society, 40–49.

Charles W. Bachman and Manilal Daya. 1977. The role concept in data models. In *VLDB*. IEEE Computer Society, 464–476.

Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. 2013. A survey on reactive programming. *ACM Comput. Surv.* 45, 4 (2013), 52:1–52:34.

Mikaël Barbero, Frédéric Jouault, Jeff Gray, and Jean Bézivin. 2007. A Practical Approach to Model Extension. In *ECMDA-FA (LNCS)*, Vol. 4530. Springer, 32–42.

Gábor Bergmann, Csaba Debreceni, István Ráth, and Dániel Varró. 2016. Query-based access control for secure collaborative modeling using bidirectional transformations. In *MoDELS*. ACM, 351–361.

Lorenzo Bettini, Viviana Bono, and Betti Venneri. 2011. Delegation by object composition. *Sci. Comput. Program.* 76, 11 (2011), 992–1014.

Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. 2006. Model transformations? Transformation models!. In *MoDELS (LNCS)*, Vol. 4199. Springer, 440–453.

Gordon S. Blair, Nelly Bencomo, and Robert B. France. 2009. Models@run.time. *IEEE Computer* 42, 10 (2009), 22–27.

Jean-Michel Bruel, Benoit Combemale, Esther Guerra, Jean-Marc Jézéquel, Jörg Kienzle, Juan De Lara, Gunter Mussbacher, Eugene Syriani, and Hans Vangheluwe. 2020. Comparing and classifying model transformation reuse approaches across metamodels. *Software and System Modeling* 19, 2 (2020), 441–465.

Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2019. A feature-based survey of model view approaches. *Software and System Modeling* 18, 3 (2019), 1931–1952.

Hugo Brunelière, Jokin García, Philippe Desfray, Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou, and Jordi Cabot. 2015a. On lightweight metamodel extension to support modeling tools agility. In *ECMFA (LNCS)*, Vol. 9153. Springer, 62–74.

Hugo Brunelière, Jokin García Perez, Manuel Wimmer, and Jordi Cabot. 2015b. EMF Views: A view mechanism for integrating heterogeneous models. In *ER (LNCS)*, Vol. 9381. Springer, 317–325.

Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. 2016. View-based model-driven software development with ModelJoin. *Software and System Modeling* 15, 2 (2016), 473–496.

Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. 2004. Tool integration at the meta-model level: the Fujaba approach. *STTT* 6, 3 (2004), 203–218.

Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. 2010. Verification and validation of declarative model-to-model transformations through invariants. *Journal of Systems and Software* 83, 2 (2010), 283–302.

Jordi Cabot, Robert Clarisó, and Daniel Riera. 2014. On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software* 93 (2014), 1–23.

Jordi Cabot and Ruth Raventós. 2006. Conceptual modelling patterns for roles. *Journal on Data Semantics V* (2006), 158–184.

Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, and Mike Hinchey. 2014. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *Journal of Systems and Software* 91 (2014), 3–23.

Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. 2009. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *IEEE Computer* 42, 10 (2009), 37–43.

Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2014. Automatic analysis of open objects in dynamic language programs. In *Static Analysis (LNCS)*, Vol. 8723. Springer, 134–150.

Krzysztof Czarnecki and Simon Helsen. 2006. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45, 3 (2006), 621–646.

Krzysztof Czarnecki and Krzysztof Pietroszek. 2006. Verifying feature-based model templates against well-formedness OCL constraints. In *GPCE*. ACM, New York, NY, USA, 211–220.

Mohamed Dahchour, Alain Pirotte, and Esteban Zimányi. 2004. A role model and its metaclass implementation. *Inf. Syst.* 29, 3 (2004), 235–270.

Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2007. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.* 376, 3 (2007), 139–163.

Juan de Lara and Esther Guerra. 2010. Deep meta-modelling with MetaDepth. In *TOOLS (LNCS)*, Vol. 6141. Springer, 1–20.

Juan de Lara and Esther Guerra. 2017. A posteriori typing for model-driven engineering: Concepts, analysis, and applications. *ACM Trans. Softw. Eng. Methodol.* 25, 4 (2017), 31:1–31:60.

Juan de Lara and Esther Guerra. 2018. Refactoring multi-level models. *ACM Trans. Softw. Eng. Methodol.* 27, 4 (2018), 17:1–17:56.

Juan de Lara, Esther Guerra, Marsha Chechik, and Rick Salay. 2018a. Model transformation product lines. In *MoDELS*. ACM, 67–77.

Juan de Lara, Esther Guerra, Jörg Kienzle, and Yanis Hattab. 2018b. Facet-oriented modelling: open objects for model-driven engineering. In *SLE*. ACM, 147–159.

Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2013. Reusable abstractions for modeling languages. *Inf. Syst.* 38, 8 (2013), 1128–1149.

Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 12:1–12:46.

Edsger Wybe Dijkstra. 1982. In *Selected writings on computing: A personal perspective*. Springer-Verlag, 60–66.

Christophe Dony, Jacques Malenfant, and Pierre Cointe. 1992. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *OOPSLA*. ACM, 201–217.

Bastien Drouot, Fahad Rafique Golra, and Joël Champeau. 2019. A Role Modeling Based Approach for Cyber Threat Analysis. In *Model-Driven Engineering and Software Development - 7th International Conference, MODELSWARD (Communications in Computer and Information Science)*, Vol. 1161. Springer, 76–100.

Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of algebraic graph transformation*. Springer.

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 263–273.

Michalis Famelis and Marsha Chechik. 2019. Managing design-time uncertainty. *Software and System Modeling* 18, 2 (2019), 1249–1284.

Claudenir M. Fonseca, João Paulo A. Almeida, Giancarlo Guizzardi, and Victorio Albani de Carvalho. 2018. Multi-level conceptual modeling: From a formal theory to a well-founded language. In *ER (LNCS)*, Vol. 11157. Springer, 409–423.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17.

Fahad Rafique Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, and Christophe Guychard. 2016. Addressing modularity for heterogeneous multi-model systems using model federation. In *MODULARITY*. ACM, 206–211.

César González-Pérez and Brian Henderson-Sellers. 2006. A powertype-based metamodelling framework. *Software and System Modeling* 5, 1 (2006), 72–90.

Esther Guerra and Juan de Lara. 2007. Event-driven grammars: relating abstract and concrete levels of visual languages. *Software and System Modeling* 6, 3 (2007), 317–347.

Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. 2020. Property satisfiability analysis for product lines of modelling languages. *IEEE Trans. Software Eng.* In press (2020), 1–20.

Esther Guerra and Mathias Soeken. 2015. Specification-driven model transformation testing. *Software and System Modeling* 14, 2 (2015), 623–644.

Clément Guy, Benoit Combemale, Steven Derrien, James Steel, and Jean-Marc Jézéquel. 2012. On model subtyping. In *ECMFA (LNCS)*, Vol. 7349. Springer, 400–415.

Terry A. Halpin. 2005. ORM 2. In *On the move to meaningful internet systems 2005: OTM 2005 Workshops (LNCS)*, Vol. 3762. Springer, 676–687.

Scott A. Hendrickson, Bryan Jett, and André van der Hoek. 2006. Layered class diagrams: Supporting the design process. In *MODELS (LNCS)*, Vol. 4199. Springer, 722–736.

Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. 2016. Feature-based classification of bidirectional transformation approaches. *Software and System Modeling* 15, 3 (2016), 907–928.

Daniel Jackson. 2006. *Software abstractions - Logic, language, and analysis*. MIT Press, London, England. See also http://alloy.mit.edu/.

Daniel Jackson and Craig Damon. 1996. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Software Eng.* 22, 7 (1996), 484–495.

Santiago P. Jácome-Guerrero and Juan de Lara. 2020. *TOTEM*: Reconciling multi-level modelling with standard two-level modelling. *Comput. Stand. Interfaces* 69 (2020), 103390.

Andrzej Jodlowski, Piotr Habela, Jacek Plodzien, and Kazimierz Subieta. 2003. Extending OO metamodels towards dynamic object roles. In *OTM Confederated International Conferences (LNCS)*, Vol. 2888. Springer, 1032–1047.

Andrzej Jodlowski, Piotr Habela, Jacek Plodzien, and Kazimierz Subieta. 2004. Dynamic object roles – Adjusting the notion for flexible modeling. In *IDEAS*. IEEE Computer Society, 449–456.

Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Sci. Comput. Program.* 72, 1-2 (2008), 31–39.

Nafiseh Kahani, Mojtaba Bagherzadeh, James R. Cordy, Juergen Dingel, and Dániel Varró. 2019. Survey and classification of model transformation tools. *Software and System Modeling* 18, 4 (2019), 2361–2397.

Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-specific modeling - Enabling full code generation*. Wiley.

Heiko Klare, Torsten Syma, Erik Burger, and Ralf H. Reussner. 2019. A categorization of interoperability issues in networks of transformations. *Journal of Object Technology* 18, 3 (2019), 4:1–20.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006a. The Epsilon Object Language (EOL). In *ECMDA-FA (LNCS)*, Vol. 4066. Springer, 128–142.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006b. Merging models with the Epsilon Merging Language (EML). In *MoDELS (LNCS)*, Vol. 4199. Springer, 215–229.

Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2008. The Epsilon Transformation Language. In *ICMT (LNCS)*, Vol. 5063. Springer, 46–60.

Dimitrios S. Kolovos, Louis M. Rose, Nikolaos Drivalos Matragkas, Richard F. Paige, Fiona A. C. Polack, and Kiran Jude Fernandes. 2010. Constructing and navigating non-invasive model decorations. In *ICMT (LNCS)*, Vol. 6142. Springer, 138–152.

Max E. Kramer, Erik Burger, and Michael Langhammer. 2013. View-centric engineering with synchronized heterogeneous models. In *VAO*. ACM, New York, NY, USA, Article 5, 6 pages.

Mirco Kuhlmann and Martin Gogolla. 2012. From UML and OCL to relational logic and back. In *MoDELS (LNCS)*, Vol. 7590. Springer, Berlin, Heidelberg, 415–431.

Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. 2016. FRaMED: full-fledge role modeling editor (tool demo). In *SLE*. ACM, 132–136.

Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. 2015. A combined formal model for relational context-dependent roles. In *SLE*. ACM, 113–124.

Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. 2014. A metamodel family for role-based modeling and programming languages. In *SLE (LNCS)*, Vol. 8706. Springer, 141–160.

Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. 2012. EMF profiles: A lightweight extension approach for EMF models. *Journal of Object Technology* 11, 1 (2012), 1–29.

Henry Lieberman. 1986. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*. ACM, 214–223.

Fernando Macías, Adrian Rutle, Volker Stolz, Roberto Rodríguez-Echeverría, and Uwe Wolter. 2018. An approach to flexible multilevel modelling. *Enterprise Modelling and Information Systems Architectures* 13 (2018), 10:1–10:35.

Frederic Madiot and Gregoire Dupe. 2018. https://www.eclipse.org/facet/. (2018).

Ivano Malavolta, Henry Muccini, Patrizio Pelliccione, and Damien A. Tamburri. 2010. Providing architectural languages and tools interoperability through model transformation technologies. *IEEE Trans. Software Eng.* 36, 1 (2010), 119–140.

Robert C. Martin, Dirk Riehle, and Frank Buschmann. 1997. *Pattern languages of program design 3*. Addison-Wesley.

Johannes Meier, Heiko Klare, Christian Tunjic, Colin Atkinson, Erik Burger, Ralf H. Reussner, and Andreas Winter. 2019. Single underlying models for projectional, multi-view environments. In *MODELSWARD*. SciTePress, 117–128.

David Méndez-Acuña, José A. Galindo, Thomas Degueule, Benoît Combemale, and Benoit Baudry. 2016. Leveraging software product lines engineering in the development of external DSLs: A systematic literature review. *J. Computer Languages, Systems & Structures* 46 (2016), 206–235.

Thierry Millan, Hervé Leblanc, and Christian Percebois. 2017. *A dynamic type system for OCL*. Technical Report IRIT/RR-2017-02-FR. Paul Sabatier University. 1–9 pages.

Linda Northrop and Paul C. Clements. 2002. *Software product lines: Practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

OCL. 2014. http://www.omg.org/spec/OCL/. (2014).

James Odell. 1994. Power types. *JOOP* 7, 2 (1994), 8–12.

Richard F. Paige, Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos, and Fiona A. C. Polack. 2009. The design of a conceptual framework and technical infrastructure for model management language engineering. In *ICECCS*. IEEE Computer Society, Washington, DC, USA, 162–171.

Jonathan Pepin, Pascal André, J. Christian Attiogbé, and Erwan Breton. 2018. Virtual extension of meta-models with facet tools. In *MODELSWARD*. SciTePress, 59–70.

Salvador Martínez Perez, Massimo Tisi, and Rémi Douence. 2017. Reactive model transformation with ATL. *Sci. Comput. Program.* 136 (2017), 1–16.

Gilles Perrouin, Moussa Amrani, Mathieu Acher, Benoît Combemale, Axel Legay, and Pierre-Yves Schobbens. 2016. Featured model types: Towards systematic reuse in modelling language engineering. In *MiSE@ICSE*. ACM, New York, NY, USA, 1–7.

Gilles Perrouin, Gilles Vanwormhoudt, Brice Morin, Philippe Lahire, Olivier Barais, and Jean-Marc Jézéquel. 2012. Weaving variability into domain metamodels. *Software and System Modeling* 11, 3 (2012), 361–383.

Michael Pradel and Martin Odersky. 2008. Scala roles - A lightweight approach towards reusable collaborations. In *ICSOFT*. INSTICC Press, 13–20.

QVT 1.3. 2016. `http://www.omg.org/spec/QVT/`. Last accessed: March 2020. (2016).

John C. Reynolds. 1998. *Theories of programming languages*. Cambridge University Press.

Ronald L. Rivest. 1994. The RC5 encryption algorithm. In *Fast Software Encryption (LNCS)*, Vol. 1008. Springer, 86–96.

Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona Polack. 2008. The Epsilon Generation Language. In *ECMDA-FA (LNCS)*, Vol. 5095. Springer, 1–16.

Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting model transformations to product lines. In *ICSE*. ACM, New York, NY, USA, 117–128.

Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: bridging between object-oriented and functional style in reactive applications. In *MODULARITY '14*. ACM, 25–36.

Guido Salvaneschi, Alessandro Margara, and Giordano Tamburrelli. 2015. Reactive programming: A walk-through. In *ICSE*. IEEE Computer Society, 953–954.

Jesús Sánchez Cuadrado and Juan de Lara. 2018. Open meta-modelling frameworks via meta-object protocols. *Journal of Systems and Software* 145 (2018), 1–24.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2014. A component model for model transformations. *IEEE Trans. Software Eng.* 40, 11 (2014), 1042–1060.

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2014. Towards the systematic construction of domain-specific transformation languages. In *ECMFA (LNCS)*, Vol. 8569. Springer, 196–212.

Douglas C. Schmidt. 2006. Guest editor's introduction: Model-driven engineering. *Computer* 39, 2 (Feb. 2006), 25–31.

Edward Sciore. 1989. Object specialization. *ACM Trans. Inf. Syst.* 7, 2 (1989), 103–122.

Christoph Seidl, Ina Schaefer, and Uwe Aßmann. 2014. DeltaEcore – A model-based delta language generation framework. In *Modellierung (LNI)*, Vol. 225. GI, Bonn, 81–96.

Friedrich Steimann. 2000. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.* 35, 1 (2000), 83–106.

Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd edition*. Addison-Wesley Professional, Upper Saddle River, NJ.

UML 2017. UML 2.5.1 OMG specification. http://www.omg.org/spec/UML/2.5.1/. (2017).

David M. Ungar and Randall B. Smith. 1987. Self: The power of simplicity. In *OOPSLA*. ACM, 227–242.

Christian Wende, Nils Thieme, and Steffen Zschaler. 2010. A role-based approach towards modular language engineering. In *SLE (LNCS)*, Vol. 5969. Springer, 254–273.

Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt. 2009. Improving domain-specific language reuse with software product line techniques. *IEEE Software* 26, 4 (2009), 47–53.

Steffen Zschaler, Pablo Sánchez, João Pedro Santos, Mauricio Alférez, Awais Rashid, Lidia Fuentes, Ana Moreira, João Araújo, and Uirá Kulesza. 2009. VML* - A family of languages for variability management in software product lines. In *SLE (LNCS)*, Vol. 5969. Springer, 82–102.

Appendix

This appendix contains the proofs of the lemmas of Section 4.2.

Proof of Lemma 4.3. Given a faceted model M, we construct a standard model $SM = \langle H, S \cup L, feats_H \rangle$. Then, we need to show that $feats_H$ is (1) left-injective and (2) surjective.

(1) The lemma demands $F \lhd facets$ to be empty, and hence no facet has other facets. This way, $feats_H = feats \circ facets^+ = feats \circ facets$, and $H \lhd facets = facets$. The lemma also demands $facets$ to be bijective and $feats$ to be left-injective, which entails that $feats_H$ is left-injective.

(2) In faceted models, both $feats$ and $facets$ are surjective, and hence so is $feats \circ facets$.

□

Proof of Lemma 4.4. Given standard models M_1 and M_2, and a faceted model type $type = \langle type_F, type_S, type_L \rangle$ with $type_F$ total, we can build a standard model type $type = \langle type_H, type_P \rangle \colon SM_1 \to SM_2$, where SM_1 and SM_2 are built as described in the proof of Lemma 4.3.

For the "if" part, we need to show that (1) $type_H$ and $type_P$ are total functions, and (2) they commute as in Figure 8.

(1) According to Equation 3, $type_H \triangleq (({facets_2}^{-1})^+ \rhd H_2) \circ type_F \circ {facets_1}^+$. First, $type_F$ is a total function, and $facets_1$ is a bijective function. This means that $type_F \circ {facets_1}^+$ is total. Since $H_2 \lhd facets_2$ is bijective, and $F_2 \lhd facets_2$ is empty, each facet is owned by exactly a host object, and so $({facets_2}^{-1})^+ \rhd H_2$ is a bijective total function. Hence, we have that $(({facets_2}^{-1})^+ \rhd H_2) \circ type_F \circ {facets_1}^+$ is a total function from H_1 to H_2. Now, $type_F$ is a total function, and so, according to the first item of Definition 4.2, $type_P$ is defined for each feature of each facet. Since $feats_1$ is surjective, this means that $type_P$ is total.

(2) We need to show that $type_H$ and $type_P$ commute with $feats_{H_i}$ (see the left of Figure 22). Since $type_F$ is total, by the first item in Definition 4.2, square (2) in Figure 22 commutes. Now, taking into account that M_1 and M_2 are standard models, $type_H$ is constructed as ${facets_2}^{-1} \circ type_F \circ facets_1$. This is so as $F_1 \lhd facets_1$ is empty, and so $facets_1^+ = facets_1$. Similarly, $F_2 \lhd facets_2$ is empty and $H_2 \lhd facets_2$ is bijective. This means that $({facets_2}^{-1})^+ = {facets_2}^{-1}$, and ${facets_2}^{-1}$ is a function. Now, if $type_H = {facets_2}^{-1} \circ type_F \circ facets_1$, then $facets_2 \circ type_H = type_F \circ facets_1$, and square (1) in Figure 22 commutes. Finally, since both M_1 and M_2 are standard models, $feats_{H_i} = feats_i \circ facets_i$, and since the composition of two commutative squares is commutative, we have the desired compatibility.

$$
\begin{array}{ccc}
H_2 & \xrightarrow{feats_{H_2}} & P_2 \\
\uparrow type_H & = & \uparrow type_P \\
H_1 & \xrightarrow{feats_{H_1}} & P_1
\end{array}
\qquad
\begin{array}{ccccc}
H_2 & \xrightarrow{facets_2} & F_2 & \xrightarrow{feats_2} & P_2 \\
\uparrow type_H & = \ (1) & \uparrow type_F & = \ (2) & \uparrow type_P \\
H_1 & \xrightarrow{facets_1} & F_1 & \xrightarrow{feats_1} & P_2
\end{array}
$$

Fig. 22: Required compatibility condition for standard model types (left). Composing the compatibility condition (right).

For the "only if" part, let's assume that $type_F$ is not a total function. This may be because some facet f has no type, or has more than one. In the first case, the owner host

object of the facet does not receive a type through $type_H$. In the second case, it receives two. In neither case $type_H$ becomes a total function. $\Box$