Facet-Oriented Modelling

Open Objects for Model-Driven Engineering

Juan de Lara
Esther Guerra
Universidad Autónoma de Madrid
Spain

Jörg Kienzle
Yanis Hattab
McGill University
Canada

Abstract

Model-driven engineering (MDE) promotes models as the principal assets in software projects. Models are built using a modelling language whose syntax is defined by a metamodel. Hence, objects in models are typed by a metamodel class, and this typing relation is static as it is established at creation time and cannot be changed later. This way, objects in MDE are *closed* and fixed with respect to the type they conform to, the slots/properties they have, and the constraints they should obey. This hampers the reuse of model-related artefacts like model transformations, as well as the opportunistic or dynamic combination of metamodels.

To alleviate this rigidity, we propose making model objects *open* so that they can acquire or drop so-called *facets*, each one contributing a type, slots and constraints to the object. Facets are defined by regular metamodels, hence being a lightweight extension of standard metamodelling. Facet metamodels may declare usage *interfaces*, and it is possible to specify *laws* that govern how facets are to be assigned to the instances of a metamodel. In this paper, we describe our proposal, report on an implementation, and illustrate scenarios where facets have advantages over other techniques.

CCS Concepts • **Software and its engineering → Design languages**; *Semantics*; *Domain specific languages*;

Keywords Model-Driven Engineering, Metamodelling, Flexible Modelling, Role-Based Modelling, Reuse, MetaDepth

ACM Reference Format:
Juan de Lara, Esther Guerra, Jörg Kienzle, and Yanis Hattab. 2018. Facet-Oriented Modelling: Open Objects for Model-Driven Engineering. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3276604.3276610

SLE '18, November 5–6, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276610

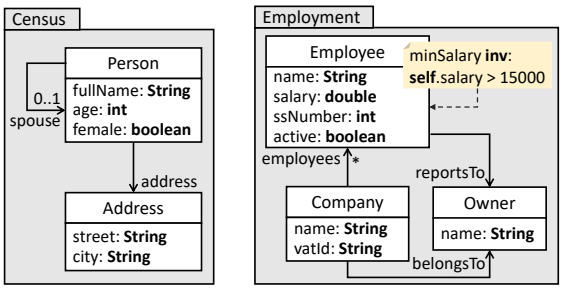
1 Introduction

Model-driven engineering (MDE) promotes the active use of models to drive the software development process [7, 38]. In MDE, models are created using a modelling language, frequently a domain-specific one. This is defined by a metamodel which declares the elements that can be used to create the models, and the constraints that the models should obey.

Classes in metamodels are used as templates for object creation, as they dictate the structural properties and well-formedness constraints that the objects of a class must comply with. When an object is created, it acquires its type, slots and constraints and these cannot change afterwards. This complicates some MDE activities. One of them is reuse as, for example, a model transformation defined over a metamodel *MM* cannot be directly applied to instances of other metamodels [8]. However, if we could assign types from *MM* to the instances of other metamodels, we could reuse the transformation as is on them. Another case is model extension, i.e., being able to increase objects with additional slots. This is useful to insert auxiliary model elements needed by a transformation, or to enable layered modelling (i.e., building models incrementally according to predefined information layers) [23]. 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 information.

There are several proposals to enable more flexible modelling, most notably *a-posteriori typing* and *role-based modelling*. In a-posteriori typing [15], objects can be added new types, but cannot acquire or drop new features dynamically. In role-based modelling [4, 31, 39], objects can acquire and drop *roles* dynamically, and such roles are typed by role types and may have attributes. Still, approaches based on roles lack declarative means to specify how the roles are to be acquired or dropped by objects, how the slots provided by a role relate to the object slots, 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 take roles as inspiration to propose a new lightweight mechanism to bring dynamism and flexibility to modelling: the *facets*. In our proposal, objects are *open* and can acquire or drop types, features and constraints, which are provided by facets. Facets are regular



(a) Census (b) Employment

Figure 1. Metamodels used in the running example.

objects, so there is no need to introduce new concepts. Different types of specifications govern when objects can take other objects as facets, favouring separation of concerns [17].

The approach is implemented on top of our metamodelling tool MetaDepth [14]. This tool is integrated with the Epsilon languages [34], which include languages for in-place and model-to-model transformation and for code generation. Our implementation permits their use taking into account the new semantics of facets. As a validation, we show how facets simplify common MDE scenarios, like reuse of model-based services and transformations, layered modelling, and multi-view modelling, comparing with other solutions.

Paper organization. Sect. 2 motivates our proposal using a running example. Sect. 3 introduces the notion of facet, and Sect. 4 defines two mechanisms to control how facets can be used: interfaces and laws. Sect. 5 reports on tool support. Sect. 6 showcases some scenarios where facets are useful. Sect. 7 discusses related research, and Sect. 8 finishes with the conclusions and prospects for future work.

2 Motivation and Running Example

We are introducing a running example to support the presentation of our approach. Assume a city hall needs a registry of the people living in the city, and for that purpose it is using the Census metamodel in Fig. 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.

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 expressed by the metamodel in Fig. 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.

For this task, one may use different strategies based on existing techniques. Fig. 2 shows a scheme of the most representative ones, which we discuss in the following.

Metamodel merging. One option is merging the metamodels in Fig. 1 statically, hence obtaining the metamodel in Fig. 2(a) or a similar one, and then migrating any existing

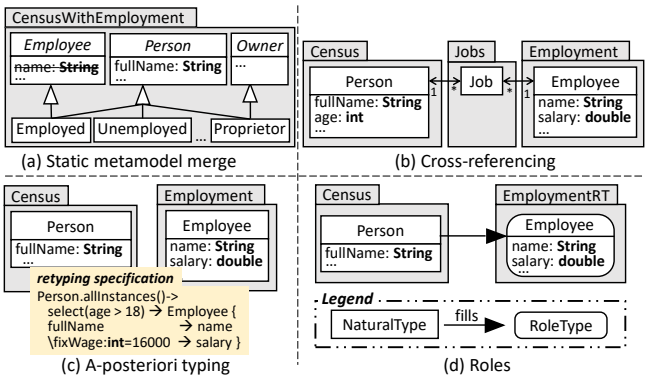


Figure 2. Alternative solutions for the running example.

Census model to this new metamodel. However, this solution has some drawbacks. First, it is not modular as it requires creating a new metamodel, as well as resolving possible issues between attributes (e.g., Employee.name is deleted and superseded by Person.fullName). Second, model migration cannot be fully automated, as 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.

Cross-referencing. Another option is to keep the metamodels in Fig. 1 separate, and use a correspondence metamodel to declare cross-references between them, as Fig. 2(b) shows. This enables dynamicity, as Employees can be created and deleted on demand, and be linked and unlinked to Person objects. However, it complicates management by adding a level of indirection, as accessing the salary of a Person or the age of an Employee requires navigating the traceability links explicitly. Instead, a solution supporting transparent access to the slots of Person and those acquired from Employee would be simpler to use. Moreover, with cross-references it becomes necessary to write a program that maintains the consistency between the models, e.g., enforcing the fullName of a Person to be the same as the name of the Employee objects it refers to. Instead, a native solution for consistency is preferable.

A-posteriori typing. Fig. 2(c) presents 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. In this approach, each slot in Employee needs to be backed on a slot of Person or on a derived attribute. In the figure, a derived attribute fixWage is created and mapped to salary, so that each Person earns the same and cannot be changed as the

attribute is derived. This is not satisfactory in the scenario, which requires adding new mutable slots to Person objects.

Roles. Fig. 2(d) sketches a solution based on roles. This solution is heavyweight and intrusive, as it requires the ad-hoc construction of role types (e.g., Employee) that mimic the metamodel types in Fig. 1(b). Some approaches [31] 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. However, 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. A more detailed analysis of roles is presented in Sect. 7.

Altogether, in the presented scenario, we would 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.
R2. Allow objects to acquire new types, slots and constraints, likely specified in other metamodels, and which become transparently accessible.
R3. Support both manual and automatic acquisition and loss of types, slots and constraints.
R4. Specification and automatic maintenance of relations (e.g., equality) between owned and acquired slots.

As none of the analysed solutions support all these requirements, next, we present a solution covering them.

3 Facets

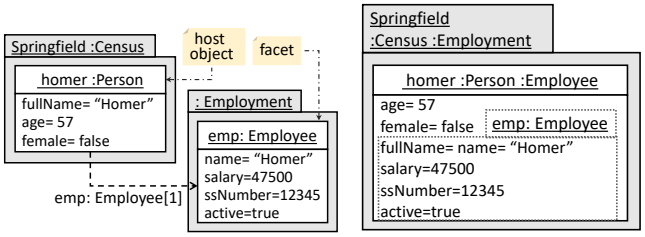
This section presents the basics of facets (Sect. 3.1), manual means to associate facets to objects (Sect. 3.2), and to view models made of faceted objects in different ways (Sect. 3.3).

3.1 The Basics of Facets

Facets are defined as follows:

A facet is an object, which becomes part of another one(s), called the host object(s), such that the slots 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.

Fig. 3(a) shows an example of a facet, which internally is an object that relates to other ones (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 Fig. 3(b). The host object acquires the type and slots provided by the facet in a transparent way. This means that, in Fig. 3(b), homer.salary is a valid feature access, and the OCL query Employee.allInstances() yields Sequence{homer}. Despite Fig. 3(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.



(a) Internal representation (b) How an object and its facets are externally perceived

Figure 3. Adding an Employee facet to the object homer.

If homer had additional facets, e.g., of type Owner, it would acquire their types and slots as well.

Facets also bring the constraints defined in their type to the host object. In the figure, the facet Employee adds constraint minSalary (cf. Fig. 1(b)) to homer. Hence, the assignment homer.salary:=13000 makes 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 Fig. 3(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, imagine that several people can share an employment, e.g., 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. Sect. 6 will illustrate other relations between fields we support, in addition to equality.

Our approach is agnostic on how host objects are created. In Fig. 3(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) [14]. Such untyped objects may be host objects as well, acquiring their type from the facets. Fig. 4 illustrates this situation, where homer is an untyped object with two facets of types Person and Employee. Similar to host objects, facets may or may not have an ontological type. While untyped facets are useful at the metamodel level, we concentrate on the usage of facets at the model level, where they are commonly typed. We call *facet metamodel* to a metamodel used to define facet types.

As facets fields are reexposed as host object fields, there may be ambiguities when field names collide in the host object and its facets, or in 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.

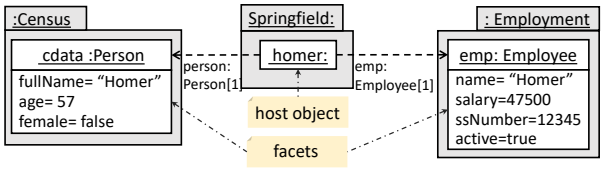


Figure 4. Host object (homer) with no creation type.

Accessing homer.salary is ambiguous as it can refer to the field of either facet. Instead, 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**, while their characteristics can be seamlessly accessed from the host objects, satisfying **R2**.

3.2 Facet Management

To add and remove facets to/from objects, we rely on queries to select the host objects to which facets are assigned. Facets may be either existing or new objects, and can be set to be shared among the selected host objects.

To simplify management, we have designed a language to manipulate facets. It is a domain-specific transformation language [12] (i.e., a constrained transformation language [13]) with the following singularities. First, we may need to refer to specific objects, and so, object identifiers may appear in queries. Second, the facets may be existing objects, may be created uniquely for each host object, or may be created and shared by several host objects. Third, we may need to specify relations among fields (like the equality relation mentioned in the previous subsection) and field value assignments.

The language has two primitives to create and delete facets, whose structure is shown in Listing 1. Both primitives support four types of queries to select host objects (line 3): by 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 ones (line 5). In the latter case, it is possible to specify a value for the fields of the added facets, or to define relations between those fields and the ones in the host object or its current facets. Newly created facets can be shared among the selected host objects with the **reuse** option (see line 5).

```
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 commands.

Primitive **removeFacet** (line 2) permits deleting facets from a set of host objects. The facets to delete can be specified by name or type (in which case, all facets of the type are removed from the objects).

As an example, Listing 2 shows the creation of a new Employee facet for the object homer. Line 1 contains the 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 [40] may require selecting objects interactively by clicking on the user interface. Line 2 creates the 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 change.

```
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 all Person objects 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 are required to satisfy complex relations. Hence, we support pattern-based selection using object tuples over which specific 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 different 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 couple.

An object can also be added more than one facet at a time. For example, Listing 5 creates two Employee facets for homer. Both facets share the fields ssNumber and active (lines 11-12), while their field name is the same as field fullName of homer.

Finally, a facet can be shared by several objects. This can be done in two ways. The first one is to assign an existing facet to another host object. For example, by using the command **addFacet** marge emp: homer.dayJob, the object marge is added homer's facet dayJob (created in Listing 5) renamed as emp. The second way is to create a new facet, and indicate that it is shared by all objects returned by a query. This is done by adding the keyword **reuse** in the **addFacet** command. For

```
addFacet homer
    dayJob: Employment.Employee with {
        name = fullName [equality]
        salary = 15000
        ssNumber = 12345
        active = true
    }
    nightJob: Employment.Employee with {
        name = fullName [equality]
        salary = 16400
        ssNumber = dayJob.ssNumber [equality]
        active = dayJob.active [equality]
    }
```

Listing 5. Making homer a moonlighter.

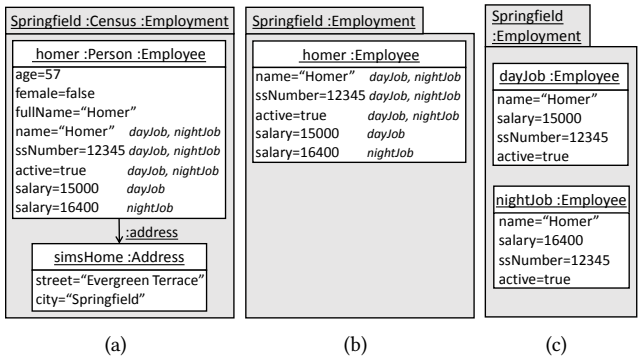


Figure 5. Three scenes of the same model: (a) total, (b) sliced by Employment, (c) granulated by Employment.

example, adding **reuse** after the command in Listing 4 makes each pair of selected objects to share the same facet.

Facet sharing, and field relations like equality, 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 way to do so, though currently restricted to equality. More complex relations may need constraint solving in the general case.

Altogether, the presented commands cover requirement **R4**, and partially **R2**, as they are a manual mechanism to specify the acquisition and loss of facets. We will present an automated mechanism for that purpose in Sect. 4.2.

3.3 Model Scenes

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. We consider three kinds of scenes: 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*. Fig. 5(a) shows a total scene of the Springfield model, where homer has the two Employee facets created in Listing 5 (dayJob and nightJob). In the figure, fields show their facets to the right.

Sliced scene. Since a model may contain objects with facets typed by different facet metamodels, we enable *slicing* a model w.r.t. a given facet metamodel. Figure 5(b) shows a sliced scene of the Springfield model w.r.t. the Employment metamodel. The slicing preserves the host objects, but it only shows the types and fields of the facet metamodel the model is sliced to. In the figure, homer is an object and not a facet, but it shows the type and fields coming from the Employment metamodel, and it lacks its original type (Person) and fields (fullName, age, female). Similarly, the sliced scene does not show the object simsHome because it has no facet from the Employment metamodel. Slicing Springfield to Census would eliminate the facet information from the model.

Granulated scene. A *granulated scene* visualizes all facets of a model that are typed by a certain facet metamodel. For this purpose, it reifies those facets as host objects, and then, it slices the model w.r.t. the facet metamodel. Fig. 5(c) shows a granulated scene of Springfield w.r.t. 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.

Models can be filtered using OCL prior to scene generation. For example, to create a scene that only considers adult persons, we can use the filter Person.allInstances()->reject(p | p<18). Scenes are visualization devices, so Employee.allInstances() returns Sequence{homer} on model Springfield regardless of the scene. To account for facets, we extend OCL with the primitive ⟨type⟩.allFacets(). This way, Employee.allFacets() returns Sequence{dayJob, nightJob} on Fig. 5.

4 Facet Laws and Interfaces

Adding facets to objects can be done in an opportunistic way, as explained in Sect. 3. While this favours agility and lightweight model extension, we also provide support for planned or systematic facet-based modelling. This is useful to control which elements can be used as facets, determine which facet types can be combined within an object, or specify declarative conditions for acquiring or losing facets, which are then enacted automatically. The proposed mechanisms are *facet interfaces* and *facet laws*, which are illustrated in Fig. 6.

The figure shows 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 which classes of FMM 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 using facets of FMM.

Facet laws are specifications of how 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 (hence fully satisfying requirement **R3**).

Our approach does not force the use of facet interfaces or laws, which remain optional. Next, we describe them.

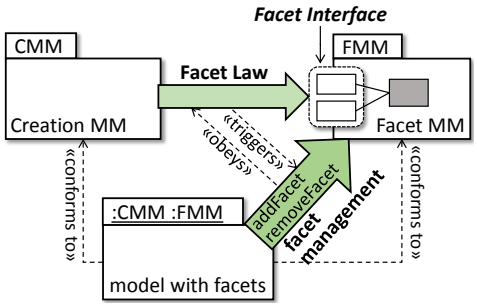


Figure 6. Facet interfaces and laws.

4.1 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 well-formedness 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 6 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 (line 3) sets of compatible classes, i.e., classes whose instances can be facets of the same host object. Classes not defined compatible are incompatible. Finally, the interface may declare additional constraints that facets and their host objects should obey (line 4).

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

Listing 6. Structure of a facet interface.

Listing 7 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 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 7. An interface for using facets of Employment.

4.2 Facet Laws

Facet laws are metamodel level specifications governing how instance models of a creation metamodel can use facets of a facet metamodel. Essentially, laws are mappings between classes in the creation and facet metamodels, to be mandatorily or optionally satisfied by the instances of those classes. They can assign a default value to the facet fields, establish field relations, and define constraints. Any facet law must respect the interface of the facet metamodel, if it exists.

Listing 8 shows the structure of facet laws. 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 laws for several creation metamodels. Sect. 6.3 will present an example of this feature. The mappings in a facet law (lines 2–3) have a similar structure to the **addFacet** command, but in addition, they specify an optionality (*must* or *may*) and can add constraints to the facets.

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

Listing 8. Structure of a facet law.

Listing 9 shows a law 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 Employment. 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 9. 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**/**removeFacet** command, every facet law that involves the metamodels of the host object and the facet is checked for consistency, banning the command if a problem is found.
- *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 9), so that they need not be repeated in every manual **addFacet** command. Thus, laws make explicit knowledge on how facets should be created explicitly.
- *To constrain facets.* By adding extra constraints to created facets (see lines 6–7 of Listing 9), 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 9, 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.

5 Tool Support

These ideas have been implemented on top of our MetaDepth tool [14]. The new version of the tool, along with all examples in the paper, is available at http://metadepth.org/mtl.

MetaDepth is a textual modelling tool that supports modelling with an arbitrary number of metalevels [1]. Hence, elements at every metalevel can be both classes (as they can be the types of lower level elements) and objects (they can be typed by other elements). This way, they are called *clabjects*, from the union of the words class and object. This makes MetaDepth level-agnostic.

```
1  Model Census {
2      Node Person{
3          name: 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 10. Census.

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

Listing 11. Springfield.

Listing 10 shows the definition of the Census metamodel in MetaDepth's syntax, while Listing 11 has an excerpt of a Census model called Springfield. The keyword to declare a metamodel (more precisely, a model with no ontological typing) is Model, and the one to create classes (clabjects with no ontological typing) is Node. Their name can be used to create instances. For example, line 1 in Listing 11 creates an instance of Census, and line 2 instantiates Address.

MetaDepth integrates with the Epsilon family of languages [34]. This permits defining in-place model transformations with EOL [26], model-to-model transformations with ETL [28], and code generators with EGL [36].

For this work, MetaDepth has been extended to support the definition of facets as a special relationship between clabjects, and by extending the semantics of (read and write) field accesses to make the access of facet fields from host clabjects transparent. To implement the equality relation between facet and host clabject fields, we use a technique that we call *mirror fields*. These are field wrappers which allow sharing the field value with other field(s). While mirror fields support equality only, this design will permit in the future to add in/out data transformation functions as decorators.

MetaDepth's typing system has been extended to incorporate into the host clabject the typing provided by its facets.

To implement the dynamic acquisition and loss of facets upon OCL conditions, we add to all clabjects of the type mentioned in the law condition what we call a *triggered constraint*. These constraints contain the OCL condition stated in the law, and create the facet when the condition is satisfied. In addition, we add another constraint with the negated OCL condition that triggers the deletion of the facet. Currently, the constraints are restricted to be "type-local", i.e., they are restricted to conditions on fields of the clabject types that take/drop the facet. For example, the condition p.age>17 in line 2 of Listing 9 is type-local (p is of type Person, which is the type of the clabjects taking the facet), while a condition like Address.allInstances()->exists(a | a.city='Springfield') (on Person objects) is not. Non type-local conditions would require an analysis to decide on which clabject type the trigger should be placed. This is left for future work.

The command set of MetaDepth has been extended to allow adding and deleting facets (**addFacet**, **removeFacet**), defining facet laws (**FacetLaws**), interfaces (**FacetInterface**), and enforcing laws (**addFacet all**).

```
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 12. EOL program showing facet dynamicity.

Altogether, facets are transparent to the model management language and can be seamlessly used within Epsilon programs. For example, assume we enforce the law in Listing 9 automatically. Then, the EOL program in Listing 12 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 the command verify, MetaDepth reports an error because the minLocalSalary constraint specified in the law (line 6 of Listing 9) fails. Line 4 makes the object p drop its facet Employee because there is no law permitting a Person to have such a

facet if the age is not greater than 17. Hence, line 5 causes an error (access to undefined field salary).

6 Application Scenarios

Next, we present three scenarios where facet-based modelling is useful. Our goal is, on the one hand, to showcase the benefits of our proposal, and on the other, to compare with other techniques for resolving the same scenarios.

6.1 Integrating Annotation Models

Several scenarios require annotating a domain model with additional information, e.g., regarding concrete syntax representation [21], uncertainty [18], access control [5], configuration of abstraction operations [16] or variability [37]. This extra information is typically 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 [18]) become available for the domain model.

Facets can be used for the same purpose. As an example, we show how to use facets to provide a graphical concrete syntax to domain models. The metamodel in Listing 13 defines simple graphical representations to be used as facets, like rectangles and circles, as subclasses of GraphicalElem. These have a position (x, y, in line 3), a label (line 4) and can be connected (line 5).

```
1  Model ConcreteSyntax {
2    abstract Node GraphicalElem {
3      x, y : int;
4      label : String;
5      linkedTo : GraphicalElem[*];
6    }
7    Node Rectangle : GraphicalElem {
8      width, height : int;
9    }
10   Node Circle : GraphicalElem {
11     radius : int;
12   }
13 }
```

Listing 13. Metamodel for visual concrete syntax.

A facet interface (omitted for space constraints) exposes Rectangle and Circle and makes them compatible, so that objects can be represented by none, either of them, or both.

Finally, Listing 14 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 require field equality, 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 Fig. 7, from a ConcreteSyntax model. ConcreteSyntax models are made of facets over objects of another model, like Springfield. This way, we obtain a textual/graphical bidirectional model synchronization for free, as for example, the graphical view is updated when a new Person is created textually, while the fields of Person objects are

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

Listing 14. Laws to add concrete syntax to Census models.

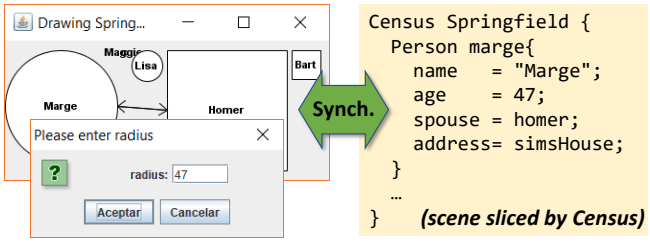


Figure 7. Concrete syntax of the Springfield model via facets.

modified whenever some graphical attribute (like the radius) changes. Facets provide this bidirectional synchronization natively, which otherwise has to be implemented ad-hoc.

Comparison with other model extension approaches. Next, we compare several ways to extend a domain model with an annotation model, for which we use the following 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.
- *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:* whether the domain objects can be recovered using the type of the annotations.
- *Bx 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 an approach fulfils this criterion if it provides native support.

Dynamicity and bx change are important features to simplify coordination of domain and annotation models. Sharing enhances expressivity. Support for typing and field access simplify the definition of transformations over annotated domain models. If unsupported, a transformation languages able to cope with several input models would be required, and the transformation becomes more complex.

Table 1 compares facets with other approaches to solve this scenario. Cross-referencing, which is the baseline, consists in having a separate annotation model that refers to the

Table 1. Approaches for model annotation

Approach	Dynamic	Sharing	Field access	Typing	Bx change
Cross-ref. (base.)	no	yes	navigation	no	no
EMF profiles [33]	limited	yes	navigation	no	no
A-posteriori [15]	limited	limited	transparent	yes	limited
Facets	yes	yes	transparent	yes	yes

objects in the domain model, similarly to Fig. 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, domain objects cannot be retrieved using the types of the annotation, there is no native support for bx change propagation.

Stereotypes can be used to emulate annotations. This approach would add a profile (the annotation metamodel) to the domain metamodel. While profiles were proposed to extend the UML metamodel [41], some proposals have adapted them to EMF metamodels [33]. Dynamicity is limited, because there is no mechanism to automatically create stereotypes, but they have to be manually added. 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. Overall, the underlying mechanism of EMF profiles are models with cross-references, presenting benefits over the baseline in terms of the specification of the correspondences, and tooling.

A-posteriori typing [15] permits typing a domain model w.r.t. 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, as 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.

Finally, facets 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.

For our example of graphical/textual concrete syntax synchronization, an approach based on cross-references would need to provide ad hoc support to bx. A-posteriori typing is not fully applicable as it does not permit mapping facet fields like x or y onto the host object fields. Finally, approaches based on stereotypes 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 [5, 16, 18, 21, 37].

6.2 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 [8]. Without proper reuse support, developers 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, lines 1–4 of Listing 15 show a simple metamodel to represent groups of elements, having an integer field. The goal of this metamodel is being able to generically define different kinds of metrics. For instance, lines 5–9 show 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 }
5 operation Group average () : Real {
6     if (elems.isEmpty()) return 0;
7     return elems.collect(quantity).sum()/
8             elems.size();
9 }
```

Listing 15. Metrics metamodel and 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 16 shows a facet law 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 9), and have a different Element facet that belongs to the field elems of the group. By defining a collection relation in line 8, every time a new adult Person is created, it is automatically added a new Element facet, which in its turn is added to field elems.

```
1  FacetLaws for People with Metrics {
2    must extend <p:Person>
3    where $p.age > 17$ with
4      ageMetric : Metrics.Element with {
5        quantity = age [equality]
6      },
7      averageAge : Metrics.Group with {
8        elems = ageMetric [collection]
9      } reuse
10 }
```

Listing 16. Laws to make Census models measurable.

After defining the laws, we can execute the program in Listing 17 over any Census model to compute the average of all objects with facet Group, without the need to adapt the original operation in Listing 15. In this case, it returns the average age of all adults. Moreover, if the age of a Person changes, or 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 17. Reusing operation average on Census models.

Table 2. Approaches for model management operation reuse

Approach	Reuse interf.	Checking type	Style	Mult. occur.	Adaptation
Model adaptation (base)	MM	no	intens.	yes	arbitrary
A-posteriori [15]	MM	syntactic	extens. intens.	no	arbitrary, bx, derived feats
Concepts [11]	MM	syntactic	extens.	no	arbitrary, derived feats/classes
Model typing [22]	MM	syntactic, semantic	extens.	no	renaming, derived feats
Facets	MM + interf.	syntactic, semantic	extens. intens.	yes	arbitrary, bx

Comparison with other reuse approaches. Next, we compare with other approaches to reusing model operations and transformations across metamodels. We base the comparison on the following subset of classification criteria identified in [8]:

- *Reuse interface:* reusable operations expose an interface for reuse which differs depending on the approach.
- *Checking type:* to validate that an operation is being correctly reused on a given model or metamodel, reuse approaches may enable static checkings (simple type-checking) or semantic checkings (e.g., well-formedness constraints).
- *Style:* the objects over which an operation is to be reused can be specified either by extension (i.e., enumerating them) or by intension (i.e., providing conditions to select them).
- *Multiple occurrences:* this is the ability to define multiple reuse contexts for an operation within a metamodel.
- *Adaptation:* to widen the reuse opportunities, reuse approaches may provide means to bridge the heterogeneities between the metamodel over which an operation is defined, and the metamodel where it is being reused.

Table 2 compares several reuse approaches. We take model adaptation as the baseline. This widely used approach 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 kind of adaptation of the source model is possible by using an intensional style. However, there are no specific checkings of the transformation correctness (thorough testing is required). Multiple occurrences are allowed, as in order to measure, e.g., 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, as if the Census model changes, we need to reexecute the adaptation transformation before the metrics operation. If the reused operation makes in-place modifications, then another transformation is needed to propagate the changes backwards.

A-posteriori typing can assign types of Metrics 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 [11] is a reuse approach inspired by generic programming. Model operations are defined over concepts (simple metamodels), which need to be bound to the concrete metamodel the operation is to be reused for. This binding induces a high-order transformation that rewrites the operation to make it applicable to the metamodel.

Model typing [22] 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 checkings expressed as OCL constraints; more expressive mappings through sharing; 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 one of the lacking features in current approaches to transformation reuse [8].

To deal with the example in Listing 17, the model adaptation approach would require transforming the Census models into Metrics models, 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 only applicable to ATL.

6.3 Multi-View Modelling

Frequently, the specification of complex systems is done by separate descriptions in different views. Having different modelling views permits separation of concerns and helps tackling the complexity of the system [3, 9]. 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 Fig. 1, while Listing 18 shows an excerpt of the metamodel for MedicalRecords, where Patients may have an insurance and go through surgeries.

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

Listing 18. Facet metamodel for health.

Facet laws can be used to specify how different view types are related and can overlap. For example, the laws in Listing 9 describe how a Census view is related to the Employment view.

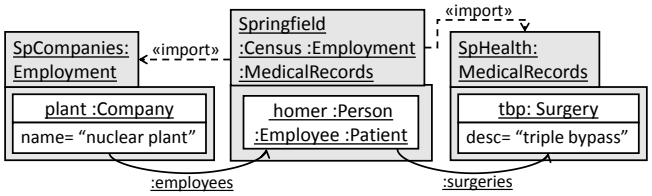


Figure 8. Models depicting different views of Springfield.

In particular, it says that, under some conditions, they overlap in classes Person and Employee.

Facet laws also permit specifying constraints involving several views, as Listing 19 shows. These laws specify how the Census and Employment view are related with MedicalRecords. The first law in lines 2–6 specifies that any Person that is also an Employee (i.e., has a facet Employee) needs to have a medical record (i.e., it should have a facet Patient). The second law in lines 7–12 states that Person objects that are not Employees need a private insurance. Finally, the third law in lines 13–18 specifies that spouses of Employees have a medical record, and get the insurance of their partner.

```
1  FacetLaws for Census, Employment with MedicalRecords {
2      must extend <p:Person> where $p.isKindOf(Employee)$
3      with healthRecord : Patient with { // job insurance
4          name = name [equality]
5          jobInsurance : $ self.active implies self.insurance $
6      }
7      may extend<p:Person> where $not p.isKindOf(Employee)$
8      with healthRecord : Patient with { // private insurance
9          name = name [equality]
10         insurance = true
11         privateInsurance : $ self.insurance=true $
12     }
13     must extend <p:Person> where
14         $p.spouse.isDefined() and p.spouse.isKindOf(Employee)$
15     with healthRecord : Patient with { // partner insurance
16         name = name [equality]
17         partnerIns: $ self.spouse.insurance implies self.insurance $
18     }
19 }
```

Listing 19. Laws for adding MedicalRecords facets to Census with Employment models.

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 Fig. 8 shows. The figure depicts three models: Springfield, SpHealth, and SpCompanies. The first model imports the other two, creating a common name space. It shows homer with facets of types Patient and Employee, where the link surgeries is due to its facet of type Patient.

Comparison with other multi-view approaches. Next, we compare with some multi-view modelling approaches. There are many proposals to handle multi-views (see [9] for a recent survey), so we just compare with some representative ones. We use the features identified in [3] as criteria:

- *View types:* in projective multi-view modelling, there is a single underlying model (SUM) for the system from which views are derived using a projection / filtering mechanism. In synthetic approaches there is no SUM, i.e., an executable system is synthetized by integrating the different 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 3. Approaches for multi-view modelling

Approach	View types	View corr.	Corr. defi.	SUM type
Central repos.	synthetic	explicit	extensional	n/a
OpenFlexo [20]	synthetic	explicit	extensional	n/a*
OSM [2]	projective	implicit	intensional	essential
Vitruvius [29]	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 [20] 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 could be used to create the illusion of an essential SUM (hence the * in Table 3).

Orthographic Software Modelling (OSM) [2] implements the classic SUM approach, where views are dynamically generated based on transformations from and to the SUM. Hence, it is a projective approach with implicit correspondences that are typically specified intensionally. The SUM in OSM is minimalistic, hence it is an essential SUM.

In Vitruvius [29], 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 unidirectional 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. Our 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.

7 Related Work

While Sect. 6 has compared several approaches to solve specific scenarios, next, we discuss additional related works.

Facets are similar to roles and fulfil most features of role languages [24, 25, 31, 32, 39]. For example, roles have properties (like facets), and objects can play several roles at the same time (host objects can acquire several facets at a time).

Comparing with specific role modelling approaches, *Lodwick* [39] unifies features of several role-based languages. However, it does not account for the instance level, nor considers model management operations as we do. ORM 2 is a well-established modelling language, but it only sees roles as relationship ends, similarly to languages such as UML or MOF. CROM [30–32] is a recent proposal to cover most features of role modelling languages. It provides a graphical editor for role types [30], but the support for role instances is still incipient. While it is a rich language with sophisticated concepts (compartment types, natural 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 we do. Moreover, facets are lightweight and do not require migrating existing metamodels into a role language.

Regarding model extension approaches, EMF-facet [19] is an Eclipse project enabling 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 of the 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 not the case for EMF-facet, which persists extended models using a specific tabular metamodel.

Model merging languages, like EML [27], permit comparing two models of the same or different metamodels, and produce a new model. While model merging has some similarities with facet-based modelling, the crucial point is that a facet and its hosts can be instances of different metamodels, and the fields of the facet are acquired by the host. This is not possible in standard model merging, as the object extension would not conform to the metamodel. Moreover, facets can be acquired and dropped dynamically, which is not supported in merging languages.

Some programming languages support incremental class definitions, like open classes (e.g., Python) and partial classes (e.g., C#). In our case, facets can work at the class or at the object level. Some programming languages have experimented with roles. For example, in [35], roles are emulated in a Scala library using a lightweight mechanism that avoids changing the language. Compared to roles in programming, our laws permit specifying declarative conditions for facet acquisition, while programming languages use imperative commands (which we also support with **addFacet**/**removeFacet**). Moreover, we handle constraints, field relations, and can configure how a facet metamodel is to be used, using interfaces.

Objects in dynamic programming languages are open: they can be added and removed slots. This has advantages, like reduced code size and increased reuse and flexibility [10], at the cost of less static assumptions and difficulty of analysis. Some modelling systems, like MetaDepth, permit adding new slots to objects at design time [14]. Instead of dealing with single slots, facets are regular objects that encapsulate slots, constraints and typing. Moreover, being defined by a metamodel facilitates defining operations over them.

8 Conclusions and Future Work

In this paper, we have proposed facets as a way to add flexibility to modelling in MDE. In contrast to standard MDE practice, facets effectively make objects *open*, so that they can acquire and drop types, slots and constraints. We have proposed explicit facet management operations for the opportunistic reuse of metamodels to create facets, as well as facet interfaces and laws for planned facet-oriented modelling. We have discussed some scenarios where facets present advantages with respect to current MDE solutions, and shown an implementation atop MetaDepth.

Facet-based modelling brings dynamicity to modelling, which opens the door to new possibilities. First, facet laws assume a model that is to be extended with facets of another metamodel. This imposes a directionality, while some applications (e.g., multi-view modelling) would benefit from a symmetrical approach, which we will develop in the future. Second, we would like to be able to analyse conflicts between the constraints in a metamodel and those in its laws and interfaces. Finally, we also plan to explore further usage scenarios for facets, like multi-abstraction modelling (where high-level elements can be seen as facets of lower-level elements) [16], models at runtime (where runtime models may change dynamically by acquiring and dropping facets) [6], or multi-level modelling (where facets may replace deep characterization mechanisms, like potency) [1].

Acknowledgements

Work partially supported by the Spanish MINECO (TIN2014-52129-R), and the Madrid region R&D program (S2013/ICE-3006), as well as NSERC Canada grant (RGPIN-2018-06610).

References

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

[2] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. 2010. 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.

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

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

[5] 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.

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

[7] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers, San Rafael, California (USA).

[8] Jean-Michel Bruel, Benoit Combemale, Esther Guerra, Jean-Marc Jézéquel, Joerg Kienzle, Juan de Lara, Gunter Mussbacher, Eugene Syriani, and Hans Vangheluwe. 2018. Model transformation reuse across metamodels: A classification and comparison of approaches. In *ICMT (LNCS)*, Vol. 10888. Springer, 1–18.

[9] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2017. A feature-based survey of model view approaches. *Software and System Modeling* (in press) (2017), 1–22.

[10] 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.

[11] 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.

[12] 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.

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

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

[15] Juan de Lara and Esther Guerra. 2017. A posteriori typing for model-driven engineering: Concepts, analysis, and applications. *ACM Transactions on Software Engineering and Methodology* 25, 4 (2017), 31:1–31:60.

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

[17] Edsger Wybe Dijkstra. 1982. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 60–66.

[18] Michalis Famelis and Marsha Chechik. 2017. Managing design-time uncertainty. *Software and System Modeling* (2017).

[19] Gregoire Dupe Frederic Madiot. 2018. https://www.eclipse.org/facet/. (2018).

[20] F. R. Golra, A. Beugnard, F. Dagnat, C. Guychard, and S. Guerin. 2016. Addressing modularity for heterogeneous multi-model systems using model federation. In *MODULARITY*. ACM, 206–211.

[21] 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.

[22] Clément Guy, Benoit Combemale, Steven Derrien, James Steel, and Jean-Marc Jézéquel. 2012. On model subtyping. In *ECMFA (LNCS)*, Vol. 7349. Springer, 400–415. https://hal.inria.fr/hal-00695034

[23] 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.

[24] 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.

[25] 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.

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

[27] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. 2006. Merging Models with the Epsilon Merging Language (EML). In *MoDELS (Lecture Notes in Computer Science)*, Vol. 4199. Springer, 215–229.

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

[29] 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.

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

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

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] D. C. Schmidt. 2006. Guest Editor's Introduction: Model-Driven Engineering. *Computer* 39, 2 (Feb. 2006), 25–31.

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

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

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