Software & Systems Modeling 14.4 (2015): 1323 – 1347

DOI: http://dx.doi.org/10.1007/s10270-013-0392-y

Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Example-driven meta-model development

Jesús J. López-Fernández*, Jesús Sánchez Cuadrado, Esther Guerra, Juan de Lara

Universidad Autónoma de Madrid (Spain),
e-mail: {Jesusj.Lopez, Jesus.Sanchez.Cuadrado, Esther.Guerra, Juan.deLara}@uam.es

Received: date / Revised version: date

Abstract The intensive use of models in Model-Driven Engineering (MDE) raises the need to develop meta-models with different aims, like the construction of textual and visual modelling languages and the specification of source and target ends of model-to-model transformations. While domain experts have the knowledge about the concepts of the domain, they usually lack the skills to build meta-models. Moreover, meta-models typically need to be tailored according to their future usage and specific implementation platform, which demands knowledge available only to engineers with great expertise in specific MDE platforms. These issues hinder a wider adoption of MDE both by domain experts and software engineers.

In order to alleviate this situation, we propose an interactive, iterative approach to meta-model construction enabling the specification of example model fragments by domain experts, with the possibility of using informal drawing tools like *Dia* or *yED*. These fragments can be annotated with hints about the *intention* or *needs* for certain elements. A meta-model is then automatically induced, which can be refactored in an interactive way, and then compiled into an *implementation* meta-model using profiles and patterns for different platforms and purposes. Our approach includes the use of a virtual assistant, which provides suggestions for improving the meta-model based on well-known refactorings, and a validation mode, enabling the validation of the meta-model by means of examples.

Key words Meta-Modelling – Domain-Specific Modelling Languages – Interactive Meta-Modelling – Meta-Model Induction – Example-Driven Modelling – Meta-Model Design Exploration – Meta-Model Validation

Send offprint requests to:
* *Present address:* Computer Science Department, Universidad Autónoma de Madrid, 28049 Madrid (Spain)

1 Introduction

Model-Driven Engineering (MDE) makes heavy use of models during the software development process. Models are usually built using Domain-Specific Modelling Languages (DSMLs) which are themselves specified through a meta-model. A DSML should contain useful, appropriate primitives and abstractions for a particular application domain. Hence, the input from domain experts and their active involvement in the meta-model development process are essential to obtain effective, useful DSMLs [34,37,38,43,53].

The usual process of meta-model construction requires first building (a part of) the meta-model which only then can be used to build instance models. Even though software engineers are used to this process, it may be counter-intuitive and difficult for non-meta-modelling experts, who may prefer drafting example models first, and then abstract those into classes and relations in a meta-model. As Oscar Nierstrasz put it, "*... in the real world, there are only objects. Classes exist only in our minds*" [45]. In this way, domain experts and final users of MDE tools are used to working with models reflecting concrete situations of their domain of expertise, but not with meta-models. Asking them to build a meta-model *before* drafting example models is often too demanding if they are not MDE experts. In general, an early exploratory phase of model construction, to understand the main concepts of the language and document the language requirements, is recommended for DSML engineering [12,37].

While MDE experts are used to work with specialized meta-modelling tools – like those provided by Eclipse EMF [52] – this is seldom the case for domain experts. These latter may find easier, more intuitive and flexible using sketching and drawing tools in the style of *PowerPoint* or *Visio* to build models and examples, than using, e.g., the EMF's tree-based editor. Moreover, once an initial version of a meta-model is built, it needs to be validated in collaboration with the domain experts.

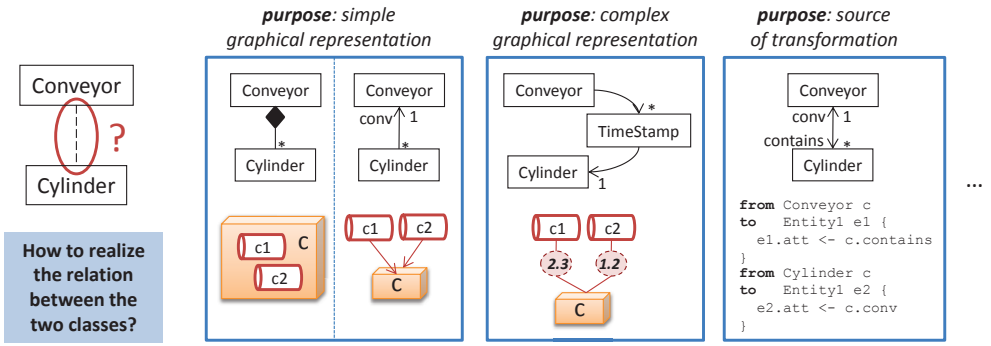

Fig. 1 Different meta-model realizations depending on its future usage.

While MDE experts are used to inspect meta-models, for domain experts a validation based on examples (again, built using sketching tools) would be more adequate, as they may lack the required expertise in conceptual modelling to fully understand a meta-model.

Another issue that makes meta-model construction cumbersome is the fact that meta-models frequently need to be fine-tuned depending on their intended use: designing a textual modelling language (e.g., with Xtext[1]), a graphical language (e.g., with GMF [33] or Eugenia [39]), or the source or target of a transformation. As illustrated in Figure 1, the particular meta-model usage may impact on its design, for instance to decide whether a connection should be implemented as a reference (e.g., for simple graphical visualization), as an intermediate class (e.g., for a more complex visualization, or to enable iterating on all connection instances), as a bidirectional association (e.g., to allow back navigation if it is used in a transformation), or as an intermediate class with composition (e.g., to enable scoping). The use of a specific technological platform, like EMF [52], has also an impact on how meta-models are actually implemented, e.g., regarding the use of composition, the need to have available a root class, and the use of references. As a consequence, the *implementation* meta-model for a particular platform may differ from the *conceptual* meta-model as elicited by domain experts. Specialized technical knowledge is required for this implementation task, hardly ever found in domain experts, which additionally has a steep learning curve.

In order to alleviate this situation, this paper presents a novel way to define meta-models and modelling environments. Its ultimate goal is to facilitate the creation of DSMLs by domain experts without proficiency in meta-modelling and MDE platforms and technologies. For this purpose, we propose an iterative process for *meta-model induction* in which model fragments are given either sketched by domain experts using drawing tools like *Dia*[2] or *yED*[3], or using a compact textual notation suitable for engineers (not necessarily meta-modelling experts). In both cases, they can annotate the *intention* of the different modelling elements. From these fragments, a meta-model is automatically induced, which can be refactored if needed. The system provides suggestions of possible improvements based on well-known refactorings [30], quality issues for conceptual schemas [2], meta-model design patterns [10] and anti-patterns [29]. Once an initial version of the meta-model is obtained, it can be validated against new model examples, and the system reports any problematic model element as well as the missing meta-model elements needed to accept the model examples. Finally, the resulting meta-model is compiled into a given technology (e.g., EMF or MetaDepth [20]), optimized for a particular purpose (e.g., visual or textual language, transformation) and a particular tool (e.g., Xtext or GMF).

This paper is an extended version of the preliminary work presented in [16]. In particular, in this extended version we provide a detailed account of additional design and domain annotations, some of which are translated into OCL constraints in the synthesized meta-model. We have also extended our tools with additional annotations and refactorings, support for an additional sketching tool and more sophisticated importing capabilities, a virtual assistant module that reports refactoring opportunities and automates their application, and a validation mode to automate testing of meta-models. The paper has also been enlarged with more extensive explanations, a more complete and challenging example, the presentation of the design of our solution, and more comprehensive related work. While in this work we concentrate on presenting the different concepts, design decisions and tool support of our approach, an empirical evaluation with our industrial partners is left for a future contribution.

Paper organization. Section 2 overviews the working scheme of our proposal. Its main steps are detailed in the following sections: specification of fragments (Section 3), meta-model induction and refactoring (Section 4), example-based meta-model validation (Section 5) and compilation of the induced meta-model for

[1] http://www.eclipse.org/Xtext/
[2] http://projects.gnome.org/dia/
[3] http://www.yworks.com/en/products_yed_about.html

different purposes and platforms (Section 6). Next, Section 7 presents tool support. Finally, Section 8 compares with related research and Section 9 ends with the conclusions and lines of future work.

2 Bottom-up Meta-modelling

Interactive development [48] promotes rapid feedback from the programming environment to the developer. Typically, a programming language provides a *shell* to write pieces of code, and the running system is updated accordingly. This permits observing the effects of the code as it is developed, and exploring different design options easily. This approach has also been regarded as a way to allow non-experts to perform simple programming tasks or to be introduced to programming, since a program is created by defining and testing small pieces of functionality that will be composed bottom-up instead of devising a complete design from the beginning. In a similar vein, example centric programming [27] promotes examples as first-class citizens in the programming process, as programs (abstractions) are iteratively and interactively developed from *concrete* examples.

Inspired by interactive and example centric programming, we propose a meta-modelling framework to facilitate the integration of end-users into the meta-modelling process, as well as permitting engineers with no meta-modelling expertise to build meta-models. The design of our framework is driven by the following requirements:

- *Bottom-up.* Whereas meta-modelling requires abstraction capabilities, the design of DSMLs demands, in addition, expert knowledge about the domain in two dimensions: horizontal and vertical [5]. The former refers to technical knowledge applicable to a range of applications (e.g., the domain of Android mobile development) and experts are developers proficient in specific implementation technologies. The vertical dimension corresponds to a particular application domain or industry (e.g., insurances) where experts are usually non-technical people. Our proposal is to let these two kinds of experts build the meta-models of DSMLs incrementally and automatically starting from example models.
  Using example models is appropriate in this context, as these two kinds of users may not be meta-modelling experts. Example models document *requirements* of the DSML to be built, provide concrete evidence on the specific use of the primitives to be supported by the DSML, and can be used for the automated derivation of its meta-model. Afterwards, the induced meta-model can be reviewed by a meta-modelling expert who can refactor some parts if needed.
  Finally, domain experts also play a crucial role in meta-model validation. Thus, we encourage their collaboration in this task by proposing an example-based validation process where end-users can feed the system with concrete examples of valid and invalid models, and the system reports whether they are correct according to the current version of the meta-model, and the reason why they are not.
- *Interactive.* A meta-model can become large, and it may address different separate concerns. In practice, its construction is an iterative process in which an initial meta-model is created, then it is tested by trying to instantiate it to create some models of interest, and whenever this is not possible, the meta-model is changed to accommodate these models [37]. The performed changes may require the detection of broken models and their manual update.
  Our proposal aims at supporting this interactive meta-model construction process. Hence, we do not advocate building a complete meta-model in one step, but the meta-model is "grown" (using the terminology of Test-Driven Development [31]) as new fragments gathering more requirements are inserted. If a new version of the meta-model breaks the conformance with existing models, the problem is reported together with possible fixes.
- *Exploratory.* The design of a meta-model is refined during its construction, and several choices are typically available for each refinement. To support the exploration of design options, we should let the developer annotate the example models with hints about the intention of the different model elements, which are then translated into some meta-model structural design decision or into additional integrity constraints. If fragments contain conflicting annotations, this is reported to the developer who can decide among the different design options. We also consider the possibility of rolling back a decision.
- *Guided by best-practices.* Since the users of this approach may not be meta-modelling experts, we consider a virtual assistant which suggests the application of meta-modelling design guidelines, best practices and refactorings that help to improve the quality of the current version of the meta-model.
- *Implementation-agnostic.* The platform used to implement a meta-model may enforce certain meta-modelling decisions (e.g., the use of compositions vs. references, or the inclusion of a root node). This knowledge is sometimes not even available to meta-modelling experts, but only to experts of the particular platform. For this reason, we postpone any decision about the target platform to a last stage. The meta-models built interactively are neutral or implementation-agnostic, and only when the meta-model design is complete, it is compiled for a specific platform.

Starting from the previous requirements, we have devised a novel process to build meta-models that is summarised in Figure 2. First, a domain expert creates one

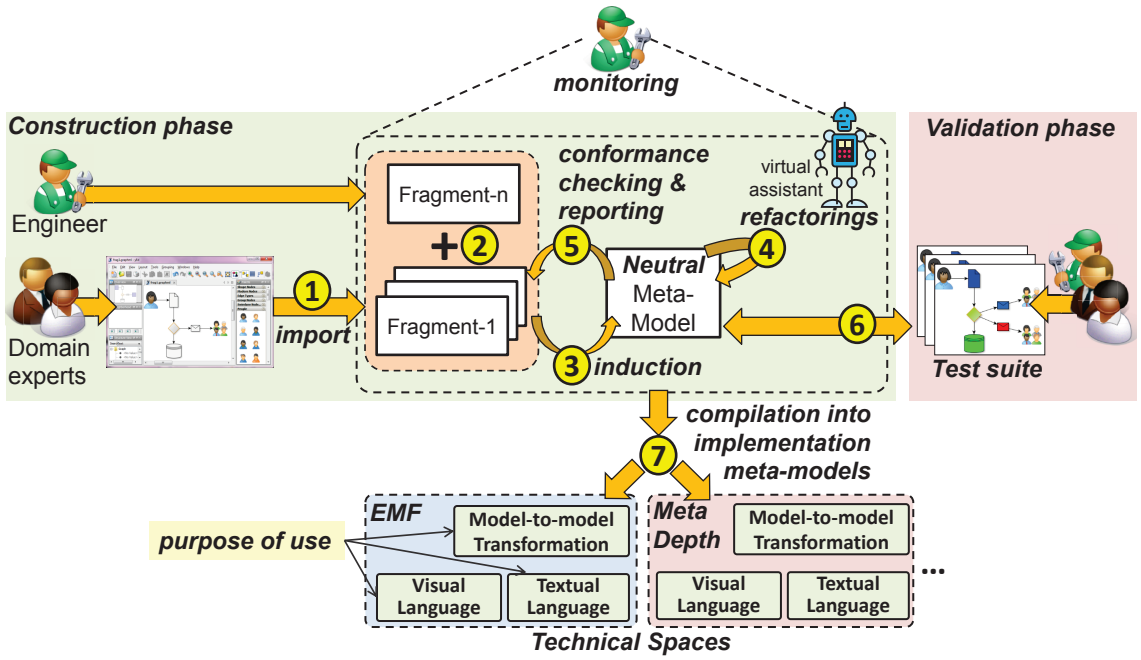

Fig. 2 Working scheme of bottom-up meta-modelling.

or more example fragments using some tool with sketching facilities, such as *Visio*, *PowerPoint*, *yED* or *Dia*. The examples do not need to be complete models, but they can concentrate on some specific concern of the language. These examples are transformed into untyped model fragments made of elements and relations (step 1). An engineer can manipulate these fragments, define new ones, and provide further annotations expressing his particular insight of certain elements in the fragments (step 2). A meta-model is automatically induced from the fragments and their annotations (step 3), and it can be visualized to gather feedback about the effect of the fragments. At this point, there are two ways to evolve the meta-model: by adding new model fragments and updating the meta-model accordingly, or by performing some refactorings suggested by the virtual assistant (step 4). In both cases, the process is monitored by the engineer, who can customise some aspects of the meta-model induction algorithm, as well as select which of the suggested refactorings should be finally applied. In addition, a checking procedure detects possible conformance issues between the new meta-model and the existing fragments, reporting potential problems and updating the fragments if possible (step 5). Once a first version of the neutral meta-model is obtained, end-users may validate it by building test cases made of example models (step 6). Thus, in the style of the *xUnit* framework [7], domain experts can feed the tool with sets of conforming and non-conforming examples to check whether the induced meta-model accepts the former and rejects the latter. The previous steps form an iterative process, so that new fragments can be added, and further validation checks performed. Finally, in step 7, the user selects a platform and purpose of use, and the neutral meta-model is compiled into an implementation one, following the specific idioms of the target technical space.

Altogether, our proposal involves two roles: domain expert and engineer. Domain experts are expected to provide background knowledge of the domain in the form of sketches, thus no technical knowledge is assumed from them. Engineers should be familiar with meta-modelling, though knowledge of concrete meta-modelling platforms is not required, as they will supervise the evolution of the neutral meta-models.

Realizing this approach poses several challenges. First of all, we foresee a process where both engineers and non-technical experts develop model fragments. Hence, they must be provided with a comprehensive set of annotations to express domain insights (mostly for domain experts) or specify design intentions (mostly for engineers). For non-technical experts, fragments are defined by sketches that have to be interpreted, for instance taking advantage of spatial relationships (e.g., containment). Secondly, the induction process is not a batch operation, but it is an interactive process that must take into account both the current version of the meta-model and the previous and new model fragments, detecting conflicts if they arise. Thirdly, a mechanism to let the users supervise the decisions of the induction algorithm has to be defined. Besides, as some users might not be meta-modelling experts, a virtual assistant needs to be available to suggest suitable refactorings. Fourth, in order to ease the participation of the domain experts in the testing phase, we propose a validation based on concrete examples (possibly made with the sketching tools) of allowed and forbidden models, together with a comprehen-

sive feedback of the detected errors. Finally, we compile meta-models for specific platforms and uses, which requires studying the requirements of the considered platforms. All these issues are discussed in Sections 3, 4, 5 and 6.

3 Definition of Model Fragments

In our approach, users provide model fragments – examples of concrete situations– from which a meta-model is induced. We call them *fragments*, because they do not need to be full-fledged models. For example, by concentrating on some aspect of interest, model fragments may miss attributes or relations, so that they do not need to be correct when evaluated as full models.

Model fragments can be specified by a domain expert, typically using a drawing tool, or by an engineer, using a more concise syntax. In both cases, fragments can include annotations about the intention of a certain part of the fragment and to guide the induction process. Fragments are used both as a documentation of specific requirements of the meta-model, and to automatically induce the meta-model, as we will see in Section 4. We normally use the term "sketch" to refer to model fragments made using a drawing tool.

As a running example, suppose we need to build an educational modelling language that will be used to plan the course syllabus, describe the structure of the courses and organize the teaching of the professors. Figure 3 shows an example model fragment as would be drawn by a domain expert (a professor) using a drawing tool (*yEd* in this case). The fragment contains a course named "Design project", with one group in the morning and another one in the afternoon, and one professor teaching each group. One of the professors is also the coordinator of the course.

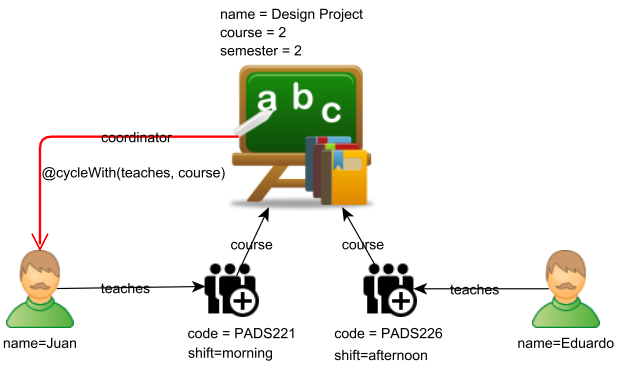

Fig. 3 A model fragment for an educational DSML.

In order to link the symbols used in the sketches with their meaning, we use another diagram serving as a kind of *legend*, or basic *reference model* for them. Figure 4 shows the legend for the example. This is a natural, technology-agnostic way for non-experts to specify the meta-model types, resembling the legend of a map, as suggested by Bézivin in [8]. This also allows using different symbols for the same concept (e.g., for professors) in order to enable more intuitive, flexible sketches.

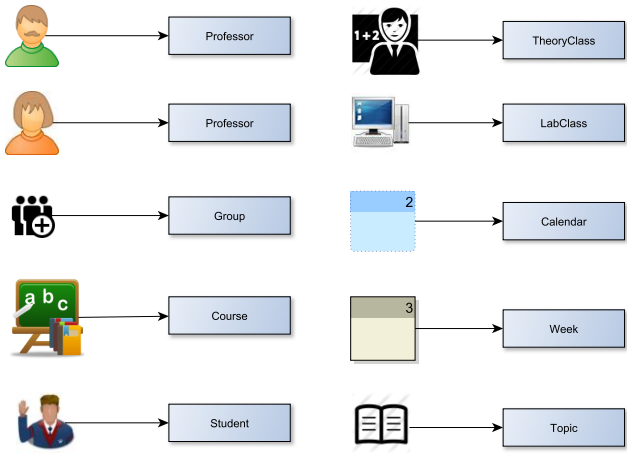

Fig. 4 Legend for the symbols used in sketches.

Listing 1 shows the sketch of Figure 3 using the textual syntax that the engineer would use. Actually, the fragment does not need to be manually written, but we have an importer for *Dia* and *yED* drawings that translates the sketches into textual fragments. The name of the types used is obtained from the legend shown in Figure 4. We will provide the technical details of this translation in Section 7.2.

```
1  fragment edu1 {
2    c : Course {
3      attr name = "Design Project"
4      attr course = 2
5      attr semester = 2
6      @cycleWith(teaches, course)
7      ref coordinator = p1
8    }
9    g1 : Group {
10     attr code = "PADS221"
11     attr shift = "morning"
12     ref course = c
13   }
14   g2 : Group {
15     attr code = "PADS226"
16     attr shift = "afternoon"
17     ref course = c
18   }
19   p1 : Professor {
20     attr name = "Juan"
21     ref teaches = g1
22   }
23   p2 : Professor {
24     attr name = "Eduardo"
25     ref teaches = g2
26   }
27 }
```

Listing 1 Model fragment in textual syntax

Both the designers and the domain experts can provide annotations to guide the induction process. These can be either *domain* or *design* annotations. Domain annotations assign a meaning or feature to certain aspects of the fragment elements, reflecting some knowl-

edge of the domain. For instance, the annotation *@cycleWith* attached to the `coordinator` reference indicates that this reference should form "a cycle" with references `teaches` and `course`, that is, the coordinator of a course should teach a group of the course. It is not necessary to repeat the same annotation for all objects of the same kind, but it is enough to annotate one of them. In the example, the annotation *@cycleWith* was added by the domain expert in the graphical fragment, but could also be added by the engineer in the textual syntax fragment, after the sketch is imported.

Table 1 shows the supported domain annotations (first column), the type of element they can annotate (second column) and their possible parameters (third column)[4]. Domain annotations are copied from the fragment to the induced meta-model, and typically produce an OCL invariant when this neutral meta-model is compiled for a specific platform (see last column of the table). The advantage of using domain annotations instead of directly OCL constraints is twofold. On the one hand, annotations are simpler to use for non meta-modelling experts, as they are higher-level than pure OCL. On the other hand, annotations get compiled into different OCL expressions depending on the properties of the annotated element (e.g., the direction of the involved references or their multiplicity), and on the particular compilation platform (as the same element can be compiled differently depending on the target platform). In the following, we explain the supported domain annotations.

The three first annotations in the table are applicable to classes. Thus, the *@unique* annotation is used to mark a certain class as a *singleton* [32] (i.e., there is at most one object of the class in each model). It can also be applied to attributes, in which case they become object identifiers. The *@container* annotation denotes that a class is a container of other classes. This has the effect of marking as *composition* the relation between the container and the containees. The *@connector* annotation marks a class as connector. It is used by our sketch importer to point out attributed associative classes derived from edges in the sketch.

As an example, Figure 5 shows a sketch expressing that students are enrolled in groups by a registration number. Listing 2 shows the result of importing the sketch, where the relation has been transformed into the class `EnrolledIn` tagged as *@connector*. As we will discuss in Section 6, this annotation has different effects depending on the target platform: while a normal class is generated in EMF, an associative class (an `Edge`) is generated in MetaDepth [20]. In the listing, the engineer has manually tagged the `studentId` attribute as *@unique*. Moreover, note that fragments do not need to include all attributes for each object (e.g., the `Group` lacks the `shift` attribute), but only the relevant ones for the given scenario.

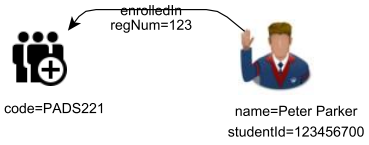

Fig. 5 Another model fragment: students enrolled in groups.

```
1  fragment edu2 {
2    s : Student {
3      attr name = "Peter Parker"
4      @unique attr studentId = 123456700
5    }
6    g : Group {
7      attr code = "PADS221"
8    }
9    @connector
10   c : EnrolledIn {
11     attr regNum = 123
12     ref student = s
13     ref group = g
14   }
15 }
```

Listing 2 Students enrolled in groups (textual syntax)

The rest of domain annotations in Table 1 denote constraints over references: *@acyclic* forbids cycles of a given reference; *@irreflexive* forbids self-loops; *@cycleWith* requires a number of references to commute, i.e., the annotated reference should form a cycle with the others; *@inverse* marks a reference as the inverse (or opposite) of another one; *@covering* indicates that a set of references with common target class is jointly surjective, i.e., any object of the target class should receive at least one of them; *@tree* indicates that a reference type spans a tree (as several disjoint trees may appear, we actually check for forests, namely, that there are no cycles, and no object has two incoming references of the type); *@subset* restricts the values held by a multivalued reference to be a subset of the values held by another one; *@xor* requires that exactly one reference of a set of references has a value, whereas the rest of references in the set should be empty or undefined; and *@nand* forbids all references in a set to have values at the same time. By providing this library of annotations we aim at facilitating the definition of commonly re-occurring meta-model constraints, even by people who are not proficient in OCL. If more complex, specific constraints were needed, they would need to be encoded by hand in the resulting meta-model.

While domain annotations make explicit expected features of some elements in the DSML, *design* annotations refer to meta-modelling design decisions that should be reflected in the meta-model generated from the fragments. These decisions can also be incorporated later by refactoring the induced meta-model, but the engineer is given the possibility to define them in advance using annotations.

[4] The "Element" and "Parameter" columns refer to the meta-model elements to which the annotation is applicable (i.e., class, reference, attribute), even if the annotations are initially included in a fragment, at the model level.

Table 1 Domain annotations for model fragments and meta-models.

Annotation	Element	Parameter	Meaning	Scheme of derived OCL invariant
@unique	class attribute	-	A given class is unique. In case of attributes, they become object identifiers.	For classes: **context** <class> **inv** unique: <class>.allInstances()−>size() <= 1
@container	class	set of classes (containees)	A given class is the container of certain elements (given by the containees parameter).	-
@connector	class	-	A given class acts as a (possibly decorated) connection between other elements.	-
@acyclic	reference	-	A given reference is acyclic.	Case 1: the upper bound of the reference is 1 **context** <ref.src> **inv** acyclic: **not** self.ref.oclIsUndefined() **implies** self−>closure(ref)−>excludes(self) Case 2: the upper bound of the reference is bigger than 1 **context** <ref.src> **inv** acyclic: self−>closure(ref)−>excludes(self)
@irreflexive	reference	-	Forbids self-loops through a reference.	Case 1: the upper bound of the reference is 1 **context** <ref.src> **inv** irreflexive: self.ref<>self Case 2: the upper bound of the reference is bigger than 1 **context** <ref.src> **inv** irreflexive: self.ref−>excludes(self)
@cycleWith	reference	reference set	If defined, a given reference must commute with a sequence of references. The sequence of references can be of any length.	Let assume a reference $ref1$ which has to commute with the sequence of references $ref2$ and $ref3$. Case 1: the upper bound of all references is 1 **context** <ref1.src> **inv** cycleWith: **if** self.ref1.oclIsUndefined() **then** true **else if** self.ref1.ref2.oclIsUndefined() **then** false **else** self.ref1.ref2.ref3 = self **endif** **endif** Case 2: the upper bound of all references is bigger than 1 **context** <ref1.src> **inv** cycleWith: self.ref1−>forAll(r1 | r1.ref2−>exists(r2 | r2.ref3−>includes(self))) Further cases are addressed using similar generation patterns.
@inverse	reference	reference	Two references are inverse of each other.	-
@covering	reference	reference set	A given set of references $\{ref_i\}$ pointing to the same class A is jointly surjective: each A object receives some reference from the set $\{ref_i\}$.	Let assume two references $ref1$ and $ref2$ with the same target. The upper bound of $ref1$ is 1, and the upper bound of $ref2$ is > 1. **context** <ref1.tar> **inv** covering: <ref1.src>.allInstances()−>exists(o | o.ref1 = self) **or** <ref2.src>.allInstances()−>exists(o | o.ref−>includes(self))
@tree	reference	-	A given reference spans a tree.	**context** <ref.src> **inv** tree: self−>closure(ref)−>excludes(self) **and** <ref.src>.allInstances()−>collect(ref)−>flatten()−>count(self) <= 1
@subset	reference	reference	The values held by a given reference are a subset of those held by another one. Both references must be owned by the same class.	Case 1: upper bound of annotated reference $(ref1) = 1$ **context** <ref1.src> **inv** subset: **not** self.ref1.oclIsUndefined() **implies** self.ref2−>includes(self.ref1) Case 2: upper bound of annotated reference $(ref1) > 1$ **context** <ref1.src> **inv** subset: self.ref2−>includesAll(self.ref1)
@xor	references	reference set	One and only one of a given set of references should have a value. All references must be owned by the same class.	Let assume two references $ref1$ and $ref2$. The upper bound of $ref1$ is 1, and the upper bound of $ref2$ is > 1. **context** <ref1.src> **inv xor**: **Sequence**{self.ref1}−>one(**not** self.oclIsUndefined()) **xor** **Sequence**{self.ref2}−>one(**not** self−>isEmpty()) For an arbitrary number of references, their name is added to the upper $Sequence$ when their upper bound is equals to 1, and to the lower $Sequence$ if their upper bound is bigger than 1.
@nand	references	reference set	A given set of references cannot all have value at the same time. All references must start from, or come into, the same class.	Let assume two references $ref1$ and $ref2$. The upper bound of $ref1$ is 1, and the upper bound of $ref2$ is > 1. Case 1: both references have the same source **context** <ref1.src> **inv** nand: self.ref1.oclIsUndefined() **or** self.ref2−>isEmpty() Case 2: both references have the same target **context** <ref1.tar> **inv** nand: (**not** <ref1.src>.allInstances()−>exists(o | o.ref1 = self)) **or** (**not** <ref2.src>.allInstances()−>exists(o | o.ref−>includes(self)))

Table 2 Design annotations for model fragments and meta-models.

Annotation	Element	Parameter	Meaning
@general	class	class	Takes every annotated class and pulls common features up to another superclass which can be specified via a parameter. The superclass can be either new or existent. If the parameter is omitted, a heuristically inferred name is provided for a new common superclass.
@general	attribute reference	-	Pulls the annotated elements up to an existing common superclass, or to a new common superclass if none exists.
@composition	reference	-	Marks the given reference as a composition.
@bidirectional	reference	-	Marks the given reference as bidirectional.

Table 2 summarizes the supported design annotations so far. The *@general* annotation specifies that a certain reference or attribute should be kept as general as possible, i.e., it should be placed as high as possible in the meta-model inheritance hierarchy. This may cause the creation of an abstract class in the meta-model, as a parent of all classes owning the reference or attribute. The annotation can also be attached to objects, and then a common parent class is created for all of them, and the maximal set of their common attributes is pulled-up to the created class. As their name suggest, the annotations *@composition* and *@bidirectional* mark a reference to be composition or bidirectional, respectively. The *@bidirectional* annotation differs from *@inverse* in that it states a design property of a single reference (that can be navigated backwards), whereas *@inverse* annotates two opposite references. *@bidirectional* may have different compilations depending on the specific meta-modelling platform (e.g., an `Edge` would get generated in MetaDepth, while two opposite references would be generated in EMF).

Altogether, annotations are a means to record an insight of the user at a given point in the running session. *Domain* annotations provide some domain knowledge, which usually results in OCL constraints attached to the resulting meta-model. *Design* annotations normally affect the structure and organization of the meta-model, and as we will see in Section 4, they are used to guide the meta-model induction process by triggering refactorings. Nonetheless, note that using annotations in fragments is optional, as the induction algorithm is able to obtain a meta-model starting from unannotated fragments (albeit probably of worse quality or less precise).

4 Bottom-up Meta-model Construction

Whenever the user enters a new fragment, the meta-model is updated accordingly to consider the new information. The annotations in the fragment are transferred to the meta-model, and this may trigger meta-model refactorings. Any conflicting information within and across fragments, like the assignment of non-compatible types for the same field, is reported to the user and automatically fixed whenever possible. Moreover, a virtual assistant provides suggestions on possible meta-model refactorings, applicable on demand.

In the following subsections, we describe our meta-model induction algorithm, how meta-model refactorings are applied, the strategy for conflict resolution, and the recommendations suggested by the virtual assistant.

4.1 The meta-model induction algorithm

Given a fragment, our algorithm proceeds by creating a new meta-class in the meta-model for each object with distinct type. If a meta-class already exists in the meta-model due to the processing of previous fragments or other objects within the same fragment, then the meta-class is not newly added. Then, for each slot in any object, a new attribute is created in the object's meta-class, if it does not exist yet. Similarly, for each reference stemming from an object, a reference type is created in its meta-class, if it does not exist. The lower bound of references is set to the minimum number of target objects connected to each object of source type, while the upper bound is set to the maximum number of target objects in the fragment. Actually, the user can configure the defaults for the lower (0 or the minimum in the fragment) and upper (unbounded or the maximum in the fragment) bounds of references. In case of selecting an unbounded maximum by default, the algorithm checks if the reference name is singular, in which case it keeps the maximum of the fragment (and the user gets the recommendation of changing the name to plural if such maximum is greater than one, see Section 4.4).

Once the meta-model has been produced, the user is allowed to decrease the lower bound and augment the upper bound of any reference induced by the algorithm. Moreover, as a consequence of processing a new fragment, the cardinality of a reference in the induced meta-model might also be relaxed: its new lower bound is set to the minimum between its current value in the meta-model and the minimum in the fragment, while its new upper bound is set to the maximum between its current value in the meta-model and the maximum in the fragment. Figure 6 shows a scheme of this situation.

If two references with the same name and stemming from objects with the same or compatible type, point to

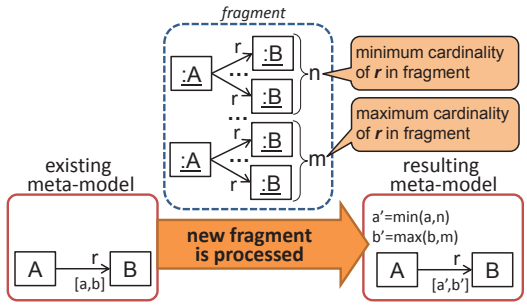

Fig. 6 Processing a reference with different cardinalities in the meta-model and a fragment.

objects of different type, our algorithm creates an abstract superclass as target of the reference type, with a subclass for the type of each target object. This situation is illustrated in Figure 7, where the new abstract class `BC` is created as parent of both `B` and `C`. Should the `B` class be abstract and the `C` object define features that are compatible with those in `B`, then `BC` would not be generated, but the new class `C` would be created as a child of `B`. The lower bound of the reference type `r` is set to `min(a, 1)` because it should accept at least one element (the one provided in the fragment), but the previous lower bound (value `a`) may be zero. As the fragment has just one reference of type `r`, the upper bound `b` of the reference is kept in the meta-model. As we will explain in Section 4.3, any automatic design decision made by the induction algorithm is reported to the user, who can change the design.

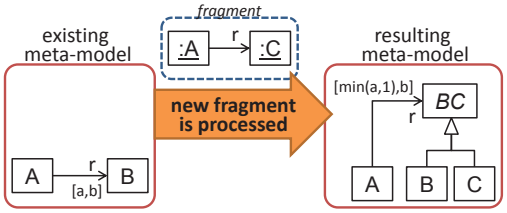

Fig. 7 Processing a reference with different target type in the meta-model and a fragment.

The algorithm also applies the previous refactoring if a fragment contains a multivalued reference holding two objects of two different classes. As an example, Figure 8 shows a fragment illustrating the structure of a course syllabus. The course has a calendar made of weeks, where only the first week is shown. In this week, two theory and one laboratory classes have been scheduled, all covering the topic "Design patterns".

Listing 3 shows the fragment once imported and translated into textual syntax. Our injector recognizes spatial relations like containment, and annotates classes `Calendar` and `Week` with one *@container* annotation each and appropriate parameters referring to the contained objects. Thus, even if the containment relations are not depicted in the sketch, appropriate relations are

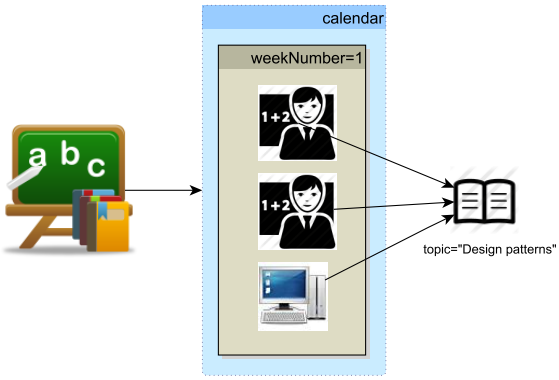

Fig. 8 Another fragment: scheduling of a course syllabus.

inferred. In addition, the engineer has manually annotated one of the `topics` references as *@general* to hint the system that this reference should be generalized (line 16). He has also annotated the `weekNumber` attribute as *@unique* to prevent different weeks with the same number (line 11).

```
1  fragment edu3 {
2    c : Course {
3      ref calendar = ca
4    }
5    @container(w)
6    ca : Calendar {
7      ref week = w
8    }
9    @container(tc1, tc2, lc)
10   w : Week {
11     @unique
12     attr weekNumber = 1
13     ref elements = tc1, tc2, lc
14   }
15   tc1 : TheoryClass {
16     @general
17     ref topics = t
18   }
19   tc2 : TheoryClass {
20     ref topics = t
21   }
22   lc : LabClass {
23     ref topics = t
24   }
25   t : Topic {
26     attr topic = "Design patterns"
27   }
28 }
```

Listing 3 Scheduling of a course syllabus (textual syntax)

Figure 9 shows the meta-model induced from this fragment, when the engineer configures the default value for the lower bound of references to be the minimum cardinality in the fragment (instead of 0), and the maximum to unbounded (taking into account the grammatical number). This is the convention that we follow from now on. Being plural, reference `elements` and `topics` receive an upper bound of *, while the rest are assigned an upper bound of 1. The lower bound of reference `topics` is 1 because all theory and lab classes refer to one topic. Since we work with references (as opposed to associations), the meta-model only keeps the cardinality of the target end of the references. As the only `Week` object in the fragment is connected to three classes, this be-

comes the lower bound of the `elements` reference. Additionally, the *@general*, *@unique* and *@container* annotations are copied from the fragment to the meta-model. In the case of *@container*, the class of the parameters is extracted. As the reference `elements` is multivalued and contains classes with different type, the algorithm creates an abstract common superclass named `Class`. The name of this new class is generated by the algorithm, using the heuristic of selecting the maximum common postfix of the names of the children classes. This decision can be overridden by the user, as we will see in Section 4.3.

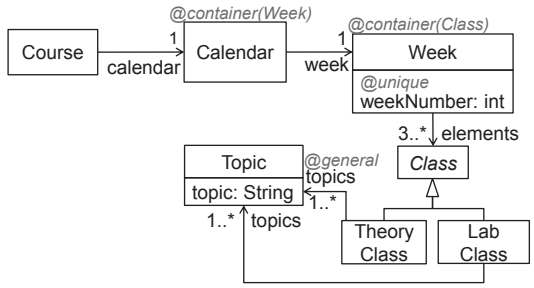

Fig. 9 Meta-model induced from the fragment in Listing 3.

4.2 Refactoring of meta-models

The design annotations are transferred from the fragments to the meta-model and may trigger refactorings in it. For example, Figure 10 shows a scheme of the refactoring triggered by the *@general* annotation applied to a reference, which is similar to the *pull-up* refactoring [30]: It pulls up the annotated attribute or reference as general as possible in the inheritance hierarchy. If the annotated attribute or reference is shared by two classes that are not related through inheritance, then an abstract, parent class is created for them so that the attribute or reference can be pulled up (i.e., Fowler's *extract superclass* refactoring [30] is applied). The target end of the pulled reference receives as lower bound the minimum of the original lower bounds, and as upper bound the maximum of the original upper bounds.

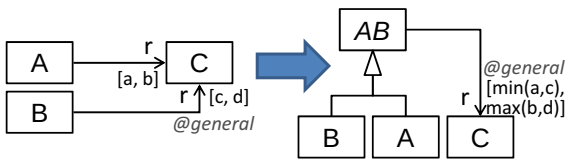

Fig. 10 Scheme of the refactoring triggered by *@general*.

Figure 11 shows the result of executing this refactoring to the meta-model in Figure 9, due to the *@general* annotation in reference `topics`. This reference is simply pulled up from both `TheoryClass` and `LabClass`, as a common parent class `Class` already exists.

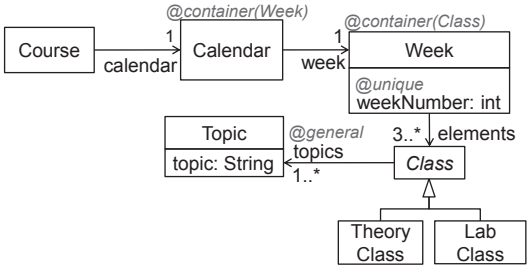

Fig. 11 Result of refactoring the meta-model in Figure 9.

4.3 Supervising decisions

Our induction process and the triggered refactorings are automated mechanisms. If there are several available design alternatives, then our algorithm takes a decision; therefore, some supervision on behalf of the user may be needed. Our aim is that the environment assists the user in refining the meta-model interactively as it is being built. To this end, our induction algorithm records the decisions taken, and presents possible alternatives to the user in the form of "open issues".

Each open issue presents one or more alternatives, each one of them associated to a refactoring. Whenever an alternative is selected, the corresponding refactoring is applied to the meta-model. This interactive approach enables non-expert users to refine a meta-model by observing the effects of their actions and following suggestions from the environment.

On the other hand, our induction algorithm is conservative as it does not break the conformance of previous fragments when the meta-model needs to be changed to accommodate new fragments; if the algorithm finds a disagreement, then it raises a conflict. However, the resolution of an open issue by means of a refactoring may break the conformance. According to [13], changes in meta-models can be classified into *non-breaking*, *breaking and resolvable*, and *breaking and unresolvable*. Our refactorings automatically update the fragments if a change is non-breaking or resolvable. For unresolvable ones, the user is asked to provide additional information or to discard the no longer conformant fragment.

We have defined two kinds of open issues: *conflict* and *automatic*, which are briefly explained next.

Conflict. The processing of new fragments may imply updating the meta-model to adjust the cardinality of existing references or add new classes, among other modifications. If a fragment contains contradictory information, then a conflict arises. For instance, there is

a conflict if the same attribute is assigned incompatible types in different objects (e.g., if a `Week` object defines an attribute `attr weekNumber=''week-1''`, and another one defines `attr weekNumber=2`, as the data type of the former is textual while the second is numerical). In this case, our algorithm chooses one of the types (e.g., String) and notifies the conflict and the alternative to the user (e.g., choosing Integer). This open issue must be resolved at some point by the designer. Changing the type of an attribute from Integer to String is an example of breaking and resolvable change (e.g., `weekNumber=2` would be automatically changed to `weekNumber=''2''`), while changing the type from String to Integer is breaking and unresolvable, and therefore requires the intervention of the user (e.g., `weekNumber=''week-1''` should be manually given a valid value). Our algorithm chooses by default an alternative that is non-breaking or, at least, resolvable.

In addition, we support the definition of conflicts and subsumption relations between annotations. In the latter case, the annotation that is weaker can be removed. An error is reported if the same element is annotated with two conflicting annotations, or if the meta-model structure is not compatible with some annotation. In particular, we detect and notify the following issues:

- a cycle of references annotated with *@containment*,
- a cycle of *@container* and *containee* objects,
- a class annotated as *@unique*, if it receives a multivalued reference,
- annotations on the same set of references where one subsumes the other: *@xor* and *@nand* (xor subsumes nand), *@acyclic* and *@tree* (tree subsumes acyclic), *@irreflexive* and *@acyclic* (acyclic subsumes irreflexive).

Automatic. These are decisions automatically taken by the induction algorithm when several alternatives exist. For instance, the name of the superclass automatically introduced for `TheoryClass` and `LabClass` is built by taking the maximal postfix that is in camel case (i.e., `Class`), or otherwise, the algorithm simply adds the prefix "General" to the concatenation of the name of the subclasses. The user is notified about this design decision, and is offered the possibility of changing the superclass' name. Similarly, the induction algorithm assigns a cardinality to the new references, and afterwards, the user can lower the minimum cardinality or increase the maximum cardinality of these references.

4.4 Recommendations

We have integrated a virtual assistant which continuously monitors the meta-model to detect places where the meta-model design can be improved and recommend solutions, based on well-known design patterns, refactorings and style guidelines. Table 3 shows the recommendations currently supported, which we categorize into *structural* and *style* suggestions. All recommendations are activated when their condition is met (third column), and if accepted by the user, they will trigger a certain meta-model refactoring (fourth column). In practice, the user may turn off the recommender, and we are currently working on fine-tuning the frequency at which recommendations are made.

The *Inline class* recommendation is given when a class `B` is referenced from another class `A` through a reference with cardinality `1..1`. Accepting the recommendation merges the two classes, i.e., the attributes and incoming/outgoing references of `B` are copied into `A`, and `B` is removed. This well-known refactoring [30], which makes sense if class `B` does not add much value by itself, results in simpler meta-models with less classes. For instance, in the meta-model of Figure 11, the assistant recommends to inline class `Calendar` into `Course`, as well as class `Week` into `Calendar`. While the former may be appropriate, the latter is not as, actually, the maximum cardinality of the reference is not correct at this stage of the iteration (i.e., a calendar may contain more than one week).

The *Pullup features* recommendation detects maximal sets of common features and references among the existing classes, and proposes either pulling the features up if a common superclass exists, or creating a common abstract superclass if the affected classes do not share a common parent. This is another well-known refactoring [30], which leads to simpler meta-models by removing duplicate fields. Technically, we use the clustering methods of Formal Concept Analysis [17] to detect sets of common features. For example, in the meta-model of Figure 9, the assistant suggests pulling up the reference `topics` to the existing superclass `Class`. Hence, the assistant frees the engineer from annotating this reference with *@general*.

The *Generalize references* recommendation proposes the creation of a common abstract superclass A for a set of classes $C = \{A_1, ..., A_n\}$ that receive a set $R = \{r_1, ..., r_n\}$ of references from another class B. In addition, a reference r from B to A is created, "merging" the reference set $\{r_1, ..., r_n\}$, which gets deleted. The cardinality of r is $[\sum_{r_i \in R} a_i, b]$, where $[a_i, b_i]$ is the cardinality of reference r_i, and $b = *$ if some $b_i = *$, else $b = \sum_{r_i \in R} b_i$. This recommendation leads to a better structured meta-model, extracting a common superclass for the $A_1, ..., A_n$ classes that reflects their commonality (all can be accessed from B). As an example, Figure 12 shows to the left a meta-model where the assistant suggests generalizing the references `theory` and `practice`. The result of applying this recommendation is shown to the right: the abstract superclass `Class` is introduced, and the original references are merged into the new reference `classes`. In addition, OCL invariants

Table 3 Recommendations for meta-model improvement.

Name	Element	Condition	Effect
Structural suggestions			
Inline class	class	A class A refers to a class B using a reference with cardinality 1..1.	Classes A and B are merged.
Pullup features	class	A set of classes define common features.	The common features are pulled up to a new or existing common superclass.
Generalize references	class and reference	A set of classes $A_1, ..., A_n$ receive references $r_1, ..., r_n$ from a class B.	A common abstract superclass A is created for $A_1, ..., A_n$, if it does not exist. References $r_1, ..., r_n$ are replaced by a new reference r from B to A, with cardinality *.
Replace class by integer	class	A featureless class without children is target of a reference.	The class is removed. An integer attribute is added to the source class of the reference.
Remove abstract class	class	A featureless abstract class has no incoming references.	The class is removed.
Naming style suggestions			
Number conflict	class attribute reference	(1) a multivalued feature has singular name, or (2) a class has plural name, or (3) a monovalued feature has plural name.	(1) suggests using a plural name, (2) suggests using a singular name, (3) suggests using a singular name or changing the multiplicity to *.
Class prefix	attribute reference	The name of a feature has the form $\langle$*owning-class-name*$\rangle X$.	Suggests renaming the feature to X.
Class camel case	class	The name of a class is not in upper camel case.	Converts the class name to upper camel case, taking care of underscores and slashes.
Feature camel case	attribute reference	The name of a feature is not in lower camel case.	Converts the feature name to lower camel case, taking care of underscores and slashes.

are generated to ensure the same cardinality of each class as in the original meta-model.

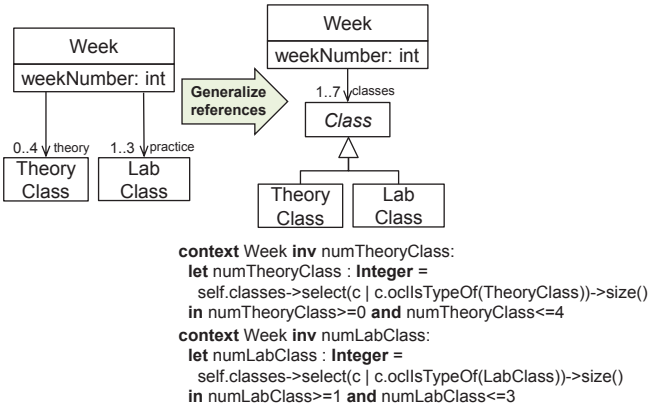

Fig. 12 *Generalize references* recommendation.

The *Replace class by integer* recommendation appears if a featureless class without children is referenced from only one class. In such a case, it is recommended to replace the class by an integer attribute in the source class of the reference, as this attribute should suffice to count the number of objects in the collection. Applying this recommendation leads to simpler meta-models, with less classes. Figure 13 illustrates this recommendation, which has been applied twice. Additional OCL invariants derived from the cardinality constraints are generated, to take care of the allowed integer values.

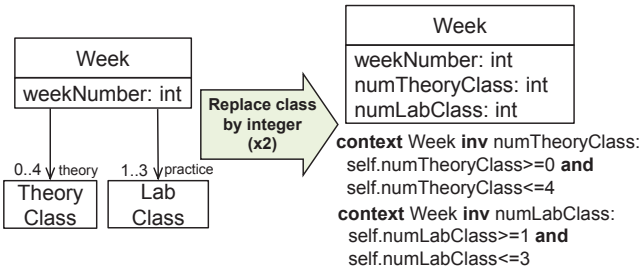

Fig. 13 *Replace class by integer* recommendation.

The *Remove abstract class* removes an intermediate featureless abstract class from an inheritance hierarchy. This class may have been created due to a generalization of some common features, which at some point have been generalized again to a higher class.

Regarding naming style suggestions, if a reference is multivalued but its name is singular, the assistant suggests changing the name to plural. If a reference is monovalued but its name is plural, the assistant suggests either changing the name to singular, or increasing the upper multiplicity to *. The default recommendation in this case can be configured by the user. For example, if we change the cardinality of `week` to 0..* in Figure 11, the assistant suggests the use of a plural name, such

as `weeks`. If an attribute name contains the name of the owning class as prefix, the assistant suggests the removal of the prefix (as recommended in [6]). As an example, the virtual assistant suggests renaming the attribute `weekNumber` of class `Week` as `number`. The resulting meta-model after applying these two recommendations is shown in Figure 14. Further suggestions take care of the capitalization of feature and class names, reflecting widely used modelling style guidelines [46].

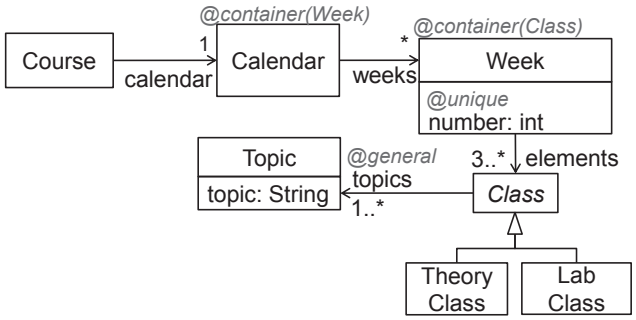

Fig. 14 Applying some naming style refactorings to the meta-model in Figure 11.

5 Example-based Validation

The collaboration of the domain experts is key to validate the meta-model and guarantee that it meets the requirements expected from the DSML. However, domain experts without knowledge of meta-modelling may find inspecting a meta-model difficult. Hence, in order to promote a more active and effective role of the domain experts in this process, we support an example-based validation of meta-models. The idea is to let the experts test the meta-model by just providing valid and invalid model examples, and return a comprehensible feedback of the test results.

More in detail, domain experts can define test suites (in the style of the xUnit framework) containing a collection of test cases. Each test case can be either a complete model or a model fragment, and can be classified as valid (the meta-model should accept it) or invalid (the meta-model should not accept it). In order to process a test suite, we use the meta-model induction algorithm presented in Section 4.1. The induction algorithm applied to a valid fragment should produce no changes in the meta-model –meaning that the meta-model accepts the fragment– whereas its application to an invalid fragment is expected to produce changes in the meta-model. For the case of complete model examples, in addition, the minimum cardinality of associations and the OCL constraints derived from domain annotations are checked (disconformities in the maximum cardinality of references are already handled by the algorithm, which triggers appropriate meta-model changes). If the test fails, an explanation of the reason is returned as feedback. A further advantage of this approach is that validation becomes more intuitive for domain experts, as it can be done using sketches, which provide a suitable concrete syntax of models. This contrasts with using the tree-based model editor provided by EMF for instantiating a meta-model.

As an example, Figure 15 shows a model example used for testing the meta-model in Figure 14. The example corresponds to the planning of a course with two weeks. If we mark this test as "model example" and "valid", the system reports the following disconformities:

1. The `unique` constraint is violated (value `1` is repeated in attribute `number`).
2. The reference `next` between topics does not exist in the meta-model.
3. The reference `subtopics` between topics does not exist in the meta-model.
4. The reference `syllabus` between a course and a topic does not exist in the meta-model.
5. The class `Course` does not define the following attributes: name, course and semester.

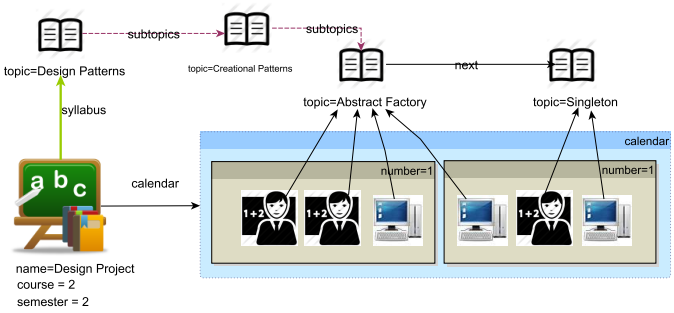

Fig. 15 Model example used for validation.

These errors are produced for two reasons. First, because the *@unique* constraint is violated (there are two `Week` objects with the same value for the attribute number). Second, because the fragments used in the meta-model induction process neglected the structure of the syllabus' topics and the course attributes. While we can use the model example as input to our induction process, a better alternative is to provide more focussed, intensional fragments that document better a certain aspect of the DSML. In this way, we sketch the fragment in Figure 16 to convey the topic structure of a syllabus, where the domain expert added the annotation *@tree* to one of the `subtopics` references, and *@acyclic* to the `next` reference.

The meta-model that results from processing this new fragment is shown in Figure 17.

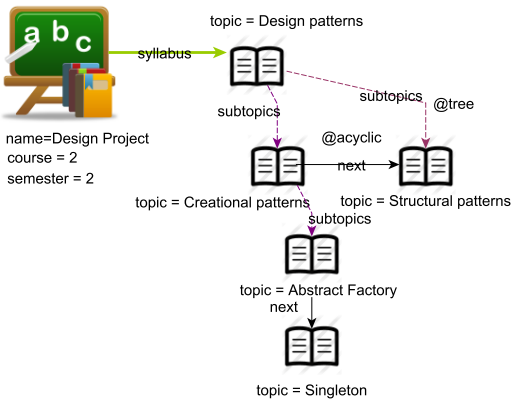

Fig. 16 Fragment describing the structure of the syllabus.

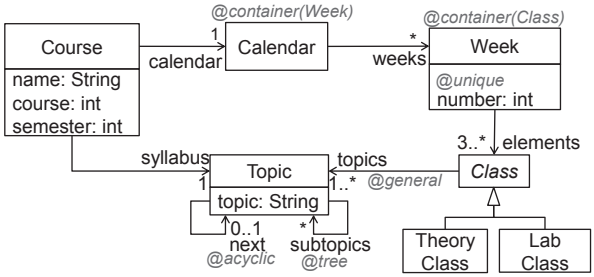

Fig. 17 Resulting meta-model.

6 Compilation for Specific Platforms

The bottom-up meta-modelling process results in a conceptual meta-model that still needs to be *implemented* in a particular platform (e.g., EMF, MetaDepth), and tweaked for a particular purpose. For example, in EMF, an extra *root* class is frequently added if the models need to be edited with the default tree editor, making heavy use of composition associations. If we aim at creating a model-to-model transformation, then we often implement references as bidirectional associations to ease the definition of navigation expressions. Therefore, we propose to define a number of transformations from the obtained neutral, conceptual meta-model into implementation ones for specific platforms and purposes.

Figure 18 shows a feature model that gathers some compilation variants from the neutral meta-model. We currently support two platforms: EMF and MetaDepth. For each one of them, one can select different profiles or purposes: transformation, visual language and textual language definition. Each platform and profile has different options that help to fine-tune the compilation.

We have also considered meta-model modularity by enabling the reuse of recurrent meta-model excerpts. A typical example is adding an existing expression language to the meta-model. At compilation time, the user selects the reused meta-model which will be integrated with the developed one. There are two modularity variants: *merge* and *extension*. With *merge*, the compiled meta-model consists of the developed meta-model plus the reused meta-model, which are merged at certain points selected by the user in a wizard (in the style of UML package merge [23]). For example, in the case of merging an expression language into our meta-model, if the expression language supports variable references, the merge points will be common meta-classes to represent the notion of "Expression" and "VariableDeclaration", and the resulting meta-model will include the whole hierarchy for expressions, as well as the common meta-class to represent variable declarations. On the contrary, with *extension* the developed meta-model just imports the reused meta-model and uses its meta-classes as types of references or by extending them. For instance, to reuse an expression language its meta-model will be imported, and there will be references to the "Expression" meta-class at each place where an expression may occur. To integrate variable references, it will be necessary to make every meta-class that acts as variable declaration inherit from "VariableDeclaration" so that it is compatible with the imported expression language meta-model. In MetaDepth, this is implemented using a dedicated meta-model extension facility [21], whereas EMF requires explicit cross-references between the meta-elements.

In both cases the user needs to select the connection points (e.g., meta-classes to represent expressions and variable declarations); the difference is how the connections are realized. In the *merge* approach, the final implementation consists of only one meta-model, which contains a copy of the elements of both meta-models. This permits implementing the rest of the artefacts of the language (e.g., concrete syntax and transformations) independently of the development of the reused meta-model, as the final meta-model is not coupled to the original reused meta-model. Instead, in the *extension* approach, any change to the reused meta-model is readily visible in the language implementation. This has the advantage that it is possible to reuse other artefacts, typically the concrete syntax. Hence, the choice of the modularity variant will be motivated by the desired degree of coupling with the reused meta-model.

Next, we enumerate the different compilations that we have considered up to now.

– **EMF platform**. This compilation produces an *Ecore* meta-model, using the rules detailed in Table 4. The uri and prefix of the meta-model (rule #1) are asked to the user by means of a wizard. Optionally, by setting the *Editable* flag to *true* (rule #7), the compilation generates a *root* class and composition associations to allow any class to be reachable from the root class via composition associations. In particular, the root is added containment references pointing to every class not referenced by another class annotated with *@container*. The compilation of domain

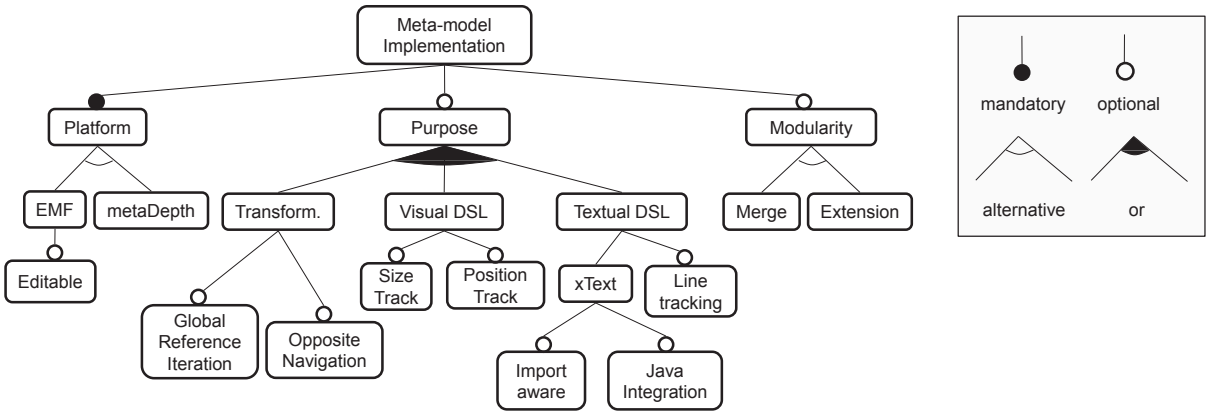

Fig. 18 Feature model for meta-model compilation.

annotations produces OCL constraints in the *Ecore* meta-model (see Table 1).

Table 4 Compilation rules for EMF.

#	**Neutral meta-model**	**Ecore**
1	Meta-model	EPackage nsPrefix = <parameter> nsURI = <parameter>
2	Class	EClass
3	Relation	EReference eType = target type lower bound = target lower bound upper bound = target upper bound
4		if annotated with *@bidirectional*: creates opposite reference in target class
5		if annotated with *@composition* or src. type annotated with *@container(tar. type)*: containment = true
6	Attribute	EAttribute eType = lookup(Attribute type)
7	Editable flag	EClass, Set(EReference)

– **MetaDepth platform**. This compilation produces a MetaDepth meta-model, which takes advantage of some special features of MetaDepth, like *Edges* to model bidirectional associations and associative classes. Table 5 shows the compilation rules. Although MetaDepth supports multi-level modelling, we limit the compilation to a standard meta-model (i.e., not multi-level), leaving the re-organization of the meta-model into several levels to future work. Please note that, as in MetaDepth all models are editable, no such feature is needed (as in EMF). The compilation of references follows a similar strategy to that for EMF (rule #3), but we generate an `Edge` for *@bidirectional* references (rule #5) and *@connector* classes (rule #6). Moreover, as MetaDepth does not natively support containment references, additional OCL constraints need to be generated to ensure that the contained objects do not belong to (i.e., are not pointed by) two different containers (rule #4). Finally, as in EMF, domain annotations may get compiled into extra OCL constraints.

Table 5 Compilation rules for MetaDepth.

#	**Neutral meta-model**	**MetaDepth**
1	Meta-model	Model with potency 1
2	Class without *@connection*	Node
3	Relation without *@bidirectional*	Reference type = target type lower bound =target lower bound upper bound =target upper bound
4		if annotated with *@composition* or src. type annotated with *@container(tar. type)*: Constraint
5	Relation with *@bidirectional*	Edge
6	Class with *@connector*	Edge
7	Attribute	Attribute type = lookup(Attribute type)
8		if annotated with *@unique* identifier = true

– **Transformation profile**. In this profile, we can configure two aspects to optimize navigation expressions. By selecting *Opposite Navigation*, selected relations become bidirectional, so that writing navigation expressions will be easier in languages making use of query expressions (like QVT). The *Global Reference Iteration* option should be selected when we foresee having to iterate over references in a global scope. A typical example is the need to apply transformation rules to inheritance relationships, which is typically hard if the child-parent relationship is only represented as a reference. In this case, an intermediate class is generated to permit the iteration. In the aforementioned example, we could generate a "Generalization" meta-class. If MetaDepth is selected as target platform, both options generate an *Edge*, which is navigable in both directions.
– **Textual language profile**. In Xtext, there is the convention of using a feature called "name" to allow cross-references to objects. Thus, any class that is target of a non-containment reference must include an attribute "name"; otherwise, it is added by the compilation. Additionally, Xtext offers the possibility

to automatically provide import facilities for textual files as well as to integrate a DSML with Java types. This requires adding certain classes and attributes to the meta-model, which is automatically done by the compiler if the variants *Import Aware* and *Java Integration* are selected. Finally, some DSMLs may require associating the line/column information to the elements (this is even required in tools like TCS), which is implemented making all classes inherit from a common *LocatedElement* class.

– **Visual language profile**. In this case, we can select whether to include in classes attributes to store the size and position of elements in the canvas.

As an example, Figure 19 shows at the top the neutral meta-model obtained by the induction process when the fragments in Figures 3, 5, 8 and 16 are considered, and the refactorings explained in Section 5 are performed. The meta-model is compiled to EMF using the transformation profile, and selecting the *Editable*, *Global Reference Iteration* and *Opposite Navigation* options. The system requests the name of the root class (`EducationalSystem` is selected), the references to be iterated (the reference `topics` is selected, to facilitate metrics that require counting the number of covered topics in every course) and those to be made bidirectional (reference `course` is selected). The resulting meta-model is shown to the bottom of Figure 19, where we have replicated class `EducationalSystem` to enhance readability. The references from classes annotated with *@container* to the classes indicated in the annotations are compiled into composition relations. This is the case for the references from `Calendar` to `Week`, and from `Week` to `Class`. The root class `EducationalSystem` is introduced, with containment relations to all classes that do not participate in any other containment relation. To facilitate the iteration over the instances of the `topics` reference, an intermediate class `CoveredTopic` is added, since Ecore does not support associations, but only references. As the `course` reference was selected for opposite navigation, a new reference `groups` is created as the opposite of `course`. Finally, some OCL invariants are generated from the *@acyclic*, *@tree*, *@cycleWith* and *@unique* annotations. For the latter case, we cannot use the standard EMF *id* property for attributes, because in that case, the checking for uniqueness is done between objects of every class having some *id* attribute, and not only among `Week` objects.

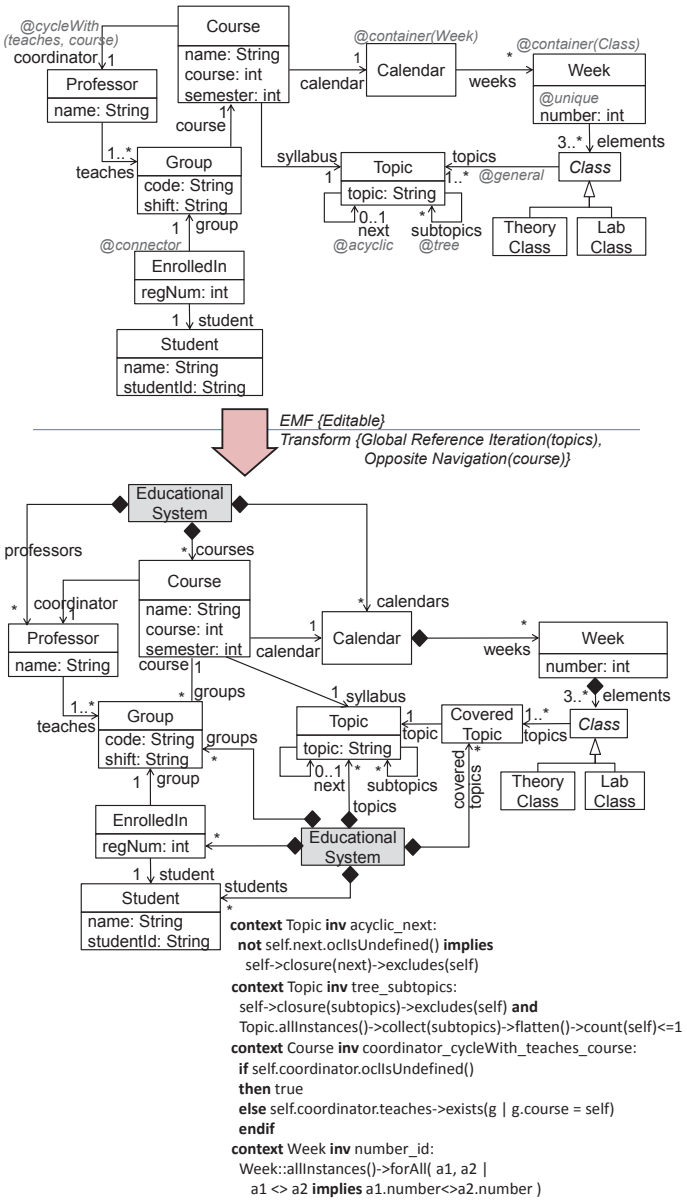

Fig. 19 Compiling to EMF for transformation.

7 Tool Support

Realizing our proposal requires specialized, integrated tool support that has to go beyond the dominant style of meta-modelling nowadays, which is mostly top-down and with limited interactivity support. To this end, we have implemented a tool for Eclipse (called *metaBUP*) that gives interactivity to our approach[5].

We present our tool in this section, which is organized as follows: We first explain the architecture of our solution, then we detail the transformation from sketches into fragments, and finally, we illustrate the different steps in the construction of a meta-model using our tool.

7.1 Architecture

Our tool has six major building blocks, which are depicted in Figure 20, and briefly summarized in the following.

[5] Available at `http://www.miso.es/tools/metaBUP.html`

1 Session manager. This module coordinates the different components at the user interface level, and persists the session state to be able to resume it in subsequent sessions. The manager also communicates with the *Fragment editor* to gather new fragments, apply the inference algorithm using the *Meta-model inferencer*, and then visualize the result using the *Meta-model visualizer*.

2 Sketch importer. The most usual way the domain expert is intended to materialize his examples is by drawing them with a general-purpose sketching tool. Thus, we have created an import facility which converts sketches drawn with *Dia* or *yED* into fragments processable by our tool. As we will detail in subsection 7.2, this import step is performed by means of two model transformations.

3 Meta-model inferencer. This component takes care of updating the meta-model when new fragments are entered, as described in Section 4. Implementation-wise, since our approach is technology-agnostic, we have created a meta-model to represent meta-models in a generic way, independently of any meta-modelling platform. We have called it *Generic Meta-model* in Figure 20. Hence, a meta-modelling session maintains a model of this kind, instead of a meta-model in a particular technology such as Ecore.

4 Meta-model validator. This module is in charge of testing whether a sequence of test cases is accepted or not by the current meta-model. For this purpose, internally, the validator clones the current meta-model, which is passed to the meta-model inferencer to obtain the list of changes that would be produced when processing the test cases. In addition, for those test cases that are model examples (as opposed to fragments), the validator checks whether they fulfil the semantics of the domain annotations as well as the lower cardinality constraints.

5 Assistant. Whenever the meta-model is updated, the assistant component analyses the new version of the meta-model to provide naming or structural change recommendations, according to the suggestions previously shown in Table 3. The assistant also takes care of applying the recommendations selected by the user. This module has an extensible architecture which makes easy the addition of new recommendations, annotations and associated refactorings in a modular way. For the *pullup feature* recommendation we have used the Colibri Java library[6] for Formal Concept Analysis.

6 Meta-model visualizer. We have implemented a visualizer for the instances of our generic meta-model using the Zest framework[7], so that the user can see the meta-model evolve.

[6] `http://code.google.com/p/colibri-java/`
[7] `http://www.eclipse.org/gef/zest`

7 Compiler. The compiler component contains the logic to compile a neutral meta-model into a platform-dependent, purpose-specific one, according to the compilation profiles presented in Section 6. Thus, it just consumes a neutral meta-model, asks the user for missing information required by the selected profile (e.g., the name of a root class), and generates an implementation meta-model accordingly.

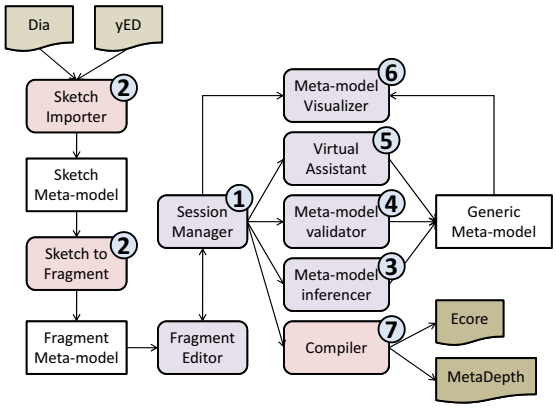

Fig. 20 Tool architecture.

Once we have seen the main elements of the architecture, we explain our support for processing sketches.

7.2 Sketch importer

To engage domain experts in the process of developing meta-models, we have developed a facility to import free-form diagrams sketched with tools like *Dia*, *yED*, *Powerpoint* or *Visio*. To support working with a variety of sketching tools, the import process is performed in two steps: first, the diagram file with the sketch is converted to a model conforming to the meta-model shown in Figure 21, which is a generic meta-model to represent diagrams; then, this model is transformed into a model fragment conformant to the meta-model shown in Figure 23, which can be processed by our tool.

The generic meta-model we use to represent the sketches read from different tools is shown in Figure 21, and encompasses the following features:

- *Type of diagram elements.* There are three types of elements in a diagram: `Symbol`, representing an element of the diagram that does not contain other elements; `Connection`, which is typically a line or an arrow connecting two elements; and `Container`, which represents a shape that may have elements "physically" inside. Thus, the `Container` metaclass captures in a uniform way the explicit containment relationship in sketches.
- *Position of elements.* The position and size of any element in the diagram is stored in attributes `x`, `y`, `width` and `height`, defined in `SketchElement`.

– *Implicit containment.* Most sketching tools allow the overlapping of elements in a diagram, but the tool may not necessarily interpret this as containment. The meta-model represents this case with the `overlapped` reference, which we calculate in a post-processing step by using the position of the elements.
– *Labels.* Elements may have `Labels` attached, for which we always store their raw text value. In addition, we distinguish three kinds of labels depending on the format of the text value: `KeyValueLabels` are labels with the form "key = value", which are useful to indicate attributes of the symbols; `AnnotationLabels` have the format "@name(param1, param2, ...)", useful to define annotations with an arbitrary number of parameters; and `PlainLabels` for labels with any other text format, which allow giving a name to symbols, connections and containers.
– *Element identification.* We use the `elemId` attribute to identify each kind of element (symbols and containers) uniquely, in order to be able to match them with the elements described in the legend. The calculation of this attribute depends on how each sketching tool identifies elements (e.g., using a unique id). Then, once an element is matched with another element in the legend, the `type` attribute is filled. If no match is found, a sensible type is automatically derived from the information provided by the sketching tool for the element. The `elemId` and `type` attributes also apply to connections, but so far we do not support assigning types to connections in the legend (e.g., depending on the colour of an arrow), and therefore we just assign default values.

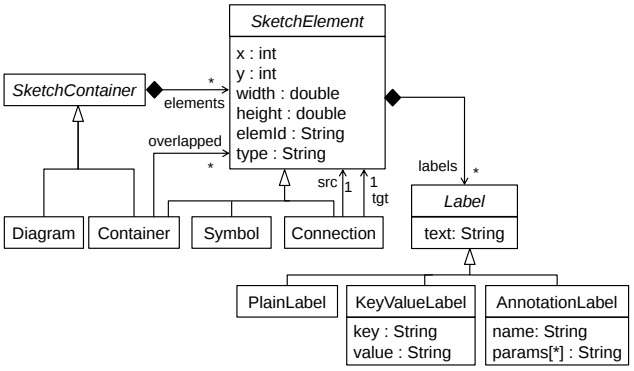

Fig. 21 Meta-model for representing sketches.

We currently support the import of diagrams created with *Dia* and *yED*. So far, the sketching meta-model has shown to be expressive enough to represent the relevant information extracted from diagrams created with both tools.

Figure 22 shows an excerpt of the model obtained after processing the sketch shown in Figure 8. The objects corresponding to symbols or containers in the sketch have been adorned with the original images to facilitate their identification. The symbol for *course* (a blackboard) is connected via a `Connection` object to the *cal* container, and the connection has a plain label conveying that the container acts as the calendar of the course. Thus, it is possible to connect symbols to containers, as well as connecting symbols (like the connection between symbols *theory1* and *topic*). The *week1* container is explicitly contained into the *cal* container, which means that the sketching tool recognized *week1* as an element inside *cal*, and our importer could fill the `elements` reference. In contrast, the sketching tool did not recognize that *theory1* is inside the *week1* container (which is clearly the case because the bounding box of the symbol overlaps with the area defined by the container). To handle this issue, our importer has an additional post-processing step to check whether overlapped elements exist, and thus fill the `overlapped` reference. This allows subsequent processing steps to give containee semantics to the overlapped elements.

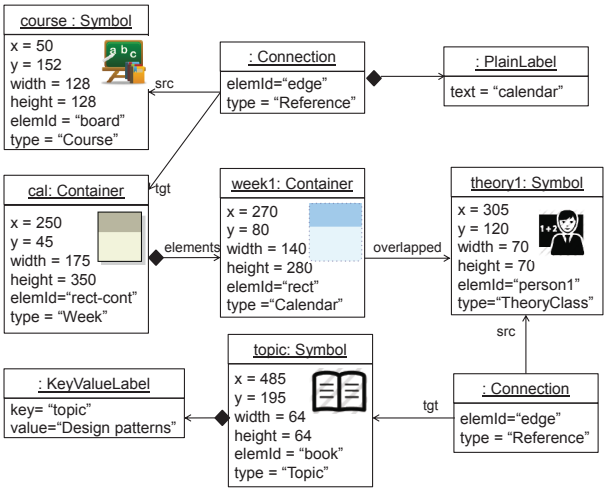

Fig. 22 Excerpt of the sketch model obtained from the sketch shown in Figure 8.

The second step consists of transforming the model representing a sketch (like that in Figure 22) into a model fragment (like that in Listing 3). The meta-model for model fragments is shown in Figure 23. A fragment is made of objects with attributes and connected by references, all of which can be annotated. Annotations can have an arbitrary number of parameters. Each parameter may have an optional name, to facilitate its understanding. We also provide some flexibility when writing parameters, allowing a set of elements to be inlined as n parameters if the annotation has only the set as parameter. In this way, these three annotations are equivalent: `@container(containee={tc1, tc2, lc})`, `@container({tc1, tc2, lc})`, `@container(tc1, tc2, lc)`.

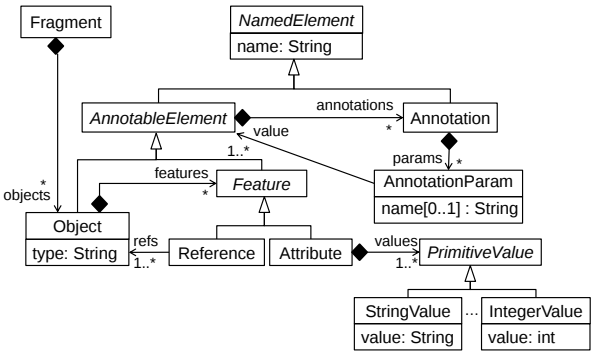

Fig. 23 Meta-model for representing fragments.

The transformation of sketches into model fragments, which is relatively straightforward, uses some heuristics to improve the quality of the generated fragment. The mapping is as follows:

- Each `Symbol` is transformed into an `Object`. In the example, *course*, *theory1* and *topic* are transformed into objects.
- Each `Container` is transformed into an `Object` annotated with *@container*. The contained objects, either explicitly or implicitly, become `containee` parameters for the container annotation. In the example, this is the case of *cal* and *week1*. According to the `elements` reference, *week1* becomes a containee of *cal*, whereas according to the `overlapped` reference, *theory1* becomes a containee of *week1*.
- A `Connection` without attributes is transformed into a `Reference` owned by the object that corresponds to the source symbol of the connection. To avoid generating many monovalued references, connections with the same name (see name resolution below) pointing to objects of the same type are grouped into a multivalued reference.
- A `Connection` with key-value labels is transformed into an object with attributes, annotated with *@connector*, and containing two references pointing to the source and target objects.
- Each `KeyValueLabel` is transformed into an `Attribute`. The value found in the sketch is parsed to detect whether it is a decimal value, an integer, a boolean value or a string. In the example, the *topic* object will have a string attribute with value "Design patterns".
- Each `AnnotationLabel` is transformed into an `Annotation`, provided that the name of the annotation label is valid (i.e., it is an annotation defined by our system). Any parameter that refers to elements in the sketch is resolved to the actual fragment elements (objects and features).
- The first `PlainLabel` (if any) of an `Object` or `Container` is used as the name for the generated object. We check for duplicate names, and add a numeric postfix if this is the case.
- The first `PlainLabel` (if any) of a `Connection` is used as the name of the generated reference or connector object. If no label is provided, we use the name of the target object to derive the name of the reference or the type of the generated connector object.

Although the sketch meta-model supports creating a connection to another connection, we do not consider this case in the mapping since neither *Dia* nor *yED* support this feature.

7.3 The tool in action

Next, we present our tool by going through an interaction example that is shown in Figure 24. First, a sketch (label 1) is built in *yED* and imported into the textual format of our tool (label 2). The tool has three tabs in the upper right panel: one with the current fragment in textual syntax (tab "Shell", which is shown in the window with label 2), one with the current induced meta-model (tab "Metamodel", which is shown in the window with label 3), and another one with the history of the inserted fragments (tab "History"). The lower panel of the tool has two tabs: one with recommendations offered by the virtual assistant (tab "Assistance"), and one with the automatic actions performed by the tool, which can be changed by the user (tab "Open issues").

When a new fragment is entered, a new meta-model is induced, or the existing one is modified to accommodate the fragment (label 3). The automatic decisions and conflicts occurring during the induction process are reported to the user in the lower panel (tab "Open issues"). In the figure, the algorithm introduced the new superclass `GeneralTheoryClassLabClass` as a generalization of others in the meta-model, and the user is prompted by a new name. This is so as the tool can be configured to create superclass names in different ways (e.g., concatenating the name of the abstracted classes prefixed by "General", as in the figure, or taking the maximal common postfix of the names). The user is informed about this decision, and may override the default name assigned to the created superclass (label 4). The window with label 5 shows the class with the new name.

These steps are performed iteratively until the meta-model contains all primitives of the domain. At any point, the user may validate the current meta-model. For this purpose, the tool contains a "Validation" tab where the user can introduce positive and negative model fragments and models, as shown in Figure 25 (the tab is to the right in the back window). The figure shows a few test cases, the first of which passed (hence it is marked with a tick), and the rest resulted in errors (hence they are marked with a cross). A dialog window offers an explanation of why a given test case failed.

The meta-model can be compiled for a particular purpose and platform, which are selected by the user, as explained in Section 6. The environment prompts a

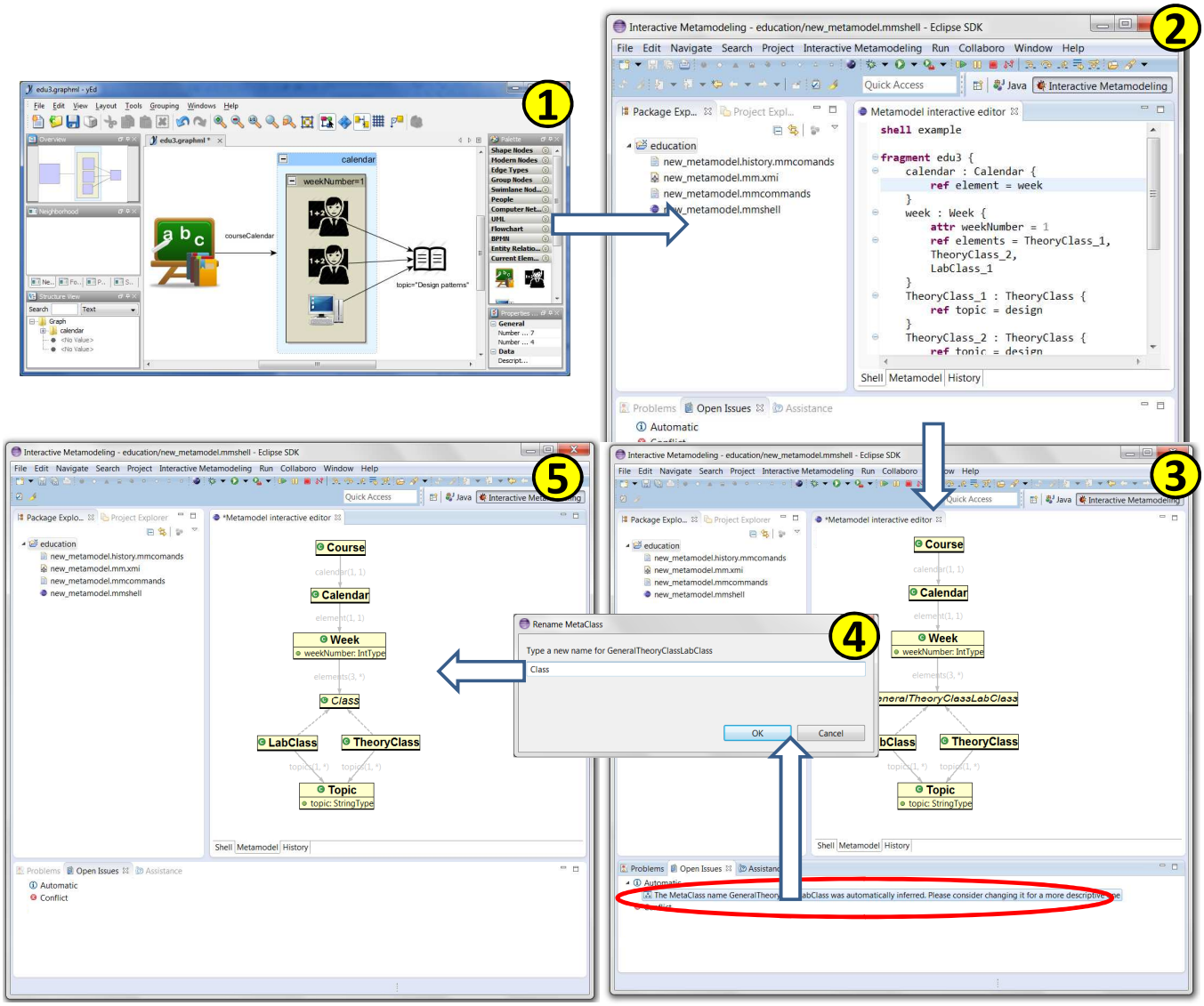

Fig. 24 Example of interaction with the tool.

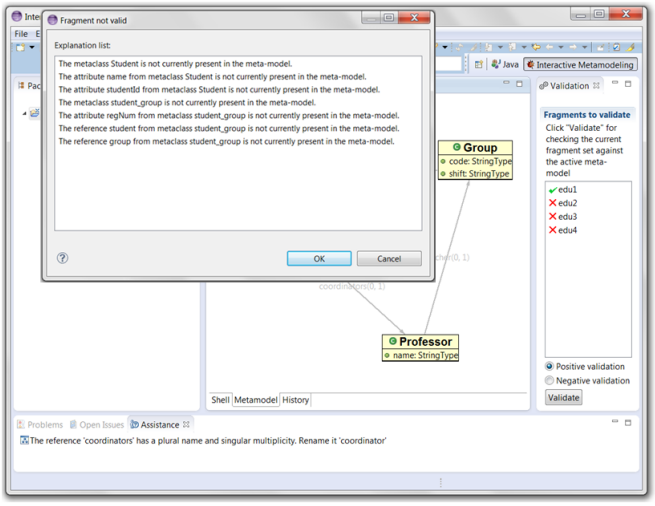

Fig. 25 Validating the meta-model.

wizard to gather any information required for the compilation (e.g., if a root class needs to be introduced when compiling into EMF, the wizard will ask its name).

8 Related Work

There are some works dealing with the inference of meta-models from models. For instance, the MARS system [36] enables the recovery of meta-models from repositories of models using grammar inference. The objective is being able to use a set of models after migrating or losing their meta-model. Actually, the induction process can be seen as a form of structural learning [40]. In contrast, our purpose is enabling the *interactive* construction of meta-models, also by domain experts.

There are a few works using test-driven development (TDD) to build meta-models iteratively. For instance, in [15], the authors attach test cases to the meta-classes

in a meta-model. Test cases are executable models written in PHP, and perform some kind of transformation like code generation. If a test case shows that a meta-model is inadequate, this must be manually modified. Similarly, in [47], the authors combine specifications and tests to guide the construction of Eiffel meta-models. Specifications are given as Eiffel contracts, whereas tests are written using the acceptance test framework for Eiffel. Another example is [49], which supports the specification of positive and negative example models from which test models for meta-model testing are generated. In our case, the meta-model is automatically induced from model fragments, and there is a greater level of interactivity. Moreover, meta-models are updated through refactorings, which simplifies their evolution and the propagation of changes to model fragments (i.e., side effects). A catalogue of meta-model refactorings, although not directly related to TDD of meta-models, is available in [44]. We provide support for many of them. Moreover, to promote the collaboration of the domain experts in the meta-model validation process, we support an example-based approach to the testing of meta-models. In the future, we plan to extend this approach to allow, e.g., the testing of meta-model invariants, using some dedicated language.

Techniques to build MDE artefacts "by example" have emerged in the last years, but it is still novel for meta-models. In the position paper [12], the authors identify some challenges to define DSMLs by demonstration. They discuss the usefulness to bridge informal drawing tools with modelling environments, as the former are the working tools of domain experts. They also recognise the difficulty for experts to manually build meta-models, and suggest an iterative process. Recently, the authors have realised their ideas in a framework where domain experts can provide model examples using a concrete syntax, from which a meta-model describing their abstract syntax is inferred [11]. While their proposal is similar to ours, we also stress that meta-models may be different depending on the target platform and usage. Hence, we support the automated induction of a *neutral* meta-model, its refactoring likely due to automatic recommendations made by a virtual assistant, its validation in collaboration with the domain experts, and different compilations into *implementation* meta-models guided through annotations and selection of configurations.

Aligned to our proposal, [4] advocates the use of examples for eliciting, modelling, verifying and validating complex business knowledge in software development projects. The authors suggest that the use of examples are a promising way to improve the quality of structural models and foster the active participation of end-users in the development process. We apply this principle to the construction of meta-models. The work in [4] was further developed in [3], considering positive and negative examples, and proposing a process that combines abstraction inference and automated example derivation. The process is supported by Clafer, a modelling tool making uniform types and instances. While our process supports abstraction inference (meta-model induction), we do not support negative examples or example derivation yet, but we will consider them in future work.

In software engineering, informal modelling and sketching is normally used in early phases of requirements elicitation [19]. Actually, it is regular practice when gathering user interface requirements in the form of mockups [51], and many mockup tools exist nowadays. In a more general setting, the work in [56] proposes a sketching tool independent of the notation which is able to transform informal sketches into more formalized models, like UML models. Our proposal can be seen as a means to leverage from DSML requirements in the form of informal sketches which are used for the construction of a meta-model. Additionally, software requirements are often specified using natural language. Several heuristics methods exist, like the noun-phrase [55], to systematically derive a conceptual model from requirements in natural language. Recently, some tools have emerged for the automated derivation (up to some degree) of class models [22]. We plan to include support for DSML requirements in natural language in future work. Conversely, some authors propose the conversion of UML/OCL conceptual schemas into natural language [9]. In our setting, this idea would be useful as a means to validate the obtained meta-model with domain experts, and also as a means to facilitate the comprehension of the constraints attached to model fragments.

Several works recommend integrating end-users in the meta-model construction process as a means to improve the quality of the resulting meta-model. For instance, in [34], the authors propose a collaborative approach named *Collaboro* to meta-model construction which involves both domain and technical experts. Their approach is supported by a DSL to represent the collaborations among stakeholders, namely change proposals, solution proposals and comments. In [35], we integrated *Collaboro* and our tool so that domain experts can collaborate by providing examples, from which change proposals are derived.

Another line of related work concerns the expressiveness of model fragments. While one could simply use object diagrams, in [41], the authors extend object diagrams with modalities to declare positive and negative model fragments and invariants (i.e., fragments that should occur in every valid diagram). Their goal is to check the consistency of a set of object diagrams, and for that purpose they use Alloy. In our case, the goal is to automatically induce a meta-model. While we consider negative fragments, they are not yet taken into account by the induction algorithm, but only for validation.

In our approach, newly introduced fragments may raise conflicts if the fragments contain contradictory information. Some application domains where the resolu-

tion of conflicts has been extensively studied are model merging [42], change propagation in software systems [28] and distributed development [14]. It is up to future work to identify how the conflicts that may arise when evolving a meta-model relate to these previous works. On the other hand, our meta-model refactorings and subsequent propagation to the model fragments can be seen as a simplified scenario of meta-model/model evolution [13].

A way to simplify and make the development of meta-models systematic is through design patterns. In [10], some design patterns for meta-models are proposed, while in [50], the requirements for meta-models are represented as use case diagrams and the meta-models are evolved by applying patterns. Related to our recommender system, in [2], the authors present a unifying approach to define *quality issues* on conceptual schemas. These issues include syntactical ones (like invalid cardinalities), naming conventions [1] (like upper camel case for the name of classes), best practices (like redundant generalizations) and basic quality issues (like concrete classes with abstract children). The authors provide a systematic way to define such issues, including their detection and resolution. In [26] an extensible framework for model recommenders is presented, where different strategies can be plugged in. We will also aim for an extension mechanism enabling the addition of new recommendations.

Our domain annotations serve as a constraint library, in the style of the predicates found in categorical sketches [24, 25]. Categorical sketches[8] are used to formalize diagrammatic notations, like the UML. In this approach, predicates are defined over an *arity shape*, which can be a graph or a family of graphs. Thus, we believe that categorical sketches could serve as a formal basis for our work.

Research in sketch recognition and understanding is also complementary to our approach [18], as handwritten, more informal sketches could be turned into fragments to feed our system. In this line, works aiming at recognizing symbols and relationships in handwritten sketches relying on user interaction [54] fit well in our approach, since the user could define the legend (i.e., the types in the meta-model) at the same time that the system is being trained.

9 Conclusions and Future Work

In this paper, we have presented a novel approach to the development of meta-models to make MDE more accessible to non-experts. For this purpose, we have proposed a bottom-up approach where a meta-model is induced from model fragments, which may be specified using informal sketching tools like *Dia* or *yED*. Sketches are transformed into a specialized textual notation that can be directly used by advanced users. Model fragments can be annotated to guide the automatic induction of the meta-model and document the intention of certain elements. The process is iterative, as fragments are added incrementally, causing updates in the meta-model, which can be refactored in the process based on the recommendations provided by a virtual assistant. To involve the domain experts in the meta-model validation process, we follow an example-based approach to the testing of meta-models. Finally, the meta-model is compiled for specific platforms and usage purposes.

Even though we allow the specification of negative fragments, these are not currently used to induce the meta-model, which is left for future work. We would like to perform an empirical evaluation of the approach with our industrial partners. Among the various aspects to be evaluated, the suitability of the library of annotations we provide is of particular interest. We also foresee the possibility of starting the process of DSML construction using a set of informal annotations, which can be formalized later. We also plan to improve the tool support. One direction is to enhance collaboration by building a web application where domain experts can sketch fragments that are automatically integrated in the environment for their refinement by an engineer. Another goal is to automatically build a visual modelling environment out of the sketched fragments. It would also be interesting to provide automated support for DSML requirements expressed in natural language, in addition to sketches. The integration of different implementation meta-models compiled from the same neutral meta-model, e.g., to support different syntaxes for a DSML, is also future work. We plan to extend our meta-model validator to allow the testing of more complex properties (e.g., meta-model invariants) using a dedicated high-level language, and provide our tool with extension mechanisms for defining new annotations, recommendations and refactorings.

Acknowledgements. We thank the referees for their detailed and useful comments. This work has been funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139), and the R&D programme of Madrid Region with project "eMadrid" (S2009/TIC-1650).

[8] Not related to the term "sketch" as we use it in this paper.

References

1. D. Aguilera, R. García-Ranea, C. Gómez, and A. Olivé. An Eclipse plugin for validating names in UML conceptual schemas. In *ER Workshops*, volume 6999 of *LNCS*, pages 323–327. Springer, 2011.
2. D. Aguilera, C. Gómez, and A. Olivé. A method for the definition and treatment of conceptual schema quality issues. In *ER*, volume 7532 of *LNCS*, pages 501–514. Springer, 2012.
3. M. Antkiewicz, K. Bak, K. Czarnecki, Z. Diskin, D. Zayan, and A. Wasowski. Example-driven modeling using clafer. In *MDEBE'2013*. CEUR, 2013.

4. K. Bak, D. Zayan, K. Czarnecki, M. Antkiewicz, Z. Diskin, A. Wasowski, and D. Rayside. Example-driven modeling: model = abstractions + examples. In *ICSE*, pages 1273–1276. IEEE / ACM, 2013.
5. C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity*, volume 1. The MIT Press, 2000.
6. R. Barker. *Case*Method: Entity Relationship Modelling.* Addison-Wesley Professional, 1990.
7. K. Beck. Simple smalltalk testing: with patterns. Technical Report 4 (2), The Smalltalk Reports, 1994.
8. J. Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2):171–188, 2005.
9. J. Cabot, R. Pau, and R. Raventós. From UML/OCL to SBVR specifications: A challenging transformation. *Inf. Syst.*, 35(4):417–440, 2010.
10. H. Cho and J. Gray. Design patterns for metamodels. In *SPLASH Workshops*, pages 25–32. ACM, 2011.
11. H. Cho, J. Gray, and E. Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *MiSE'12*, 2012.
12. H. Cho, Y. Sun, J. Gray, and J. White. Key challenges for modeling language creation by demonstration. In *ICSE'11 Workshop on Flexible Modeling Tools*, 2011.
13. A. Cicchetti, D. di Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *EDOC'08*, pages 222–231, 2008.
14. A. Cicchetti, D. di Ruscio, and A. Pierantonio. Managing model conflicts in distributed development. In *MODELS'08*, volume 5301 of *LNCS*, pages 311–325. Springer, 2008.
15. A. Cicchetti, D. di Ruscio, A. Pierantonio, and D. Kolovos. A test-driven approach for metamodel development. In *Emerging Technologies for the Evolution and Maintenance of Software Models*, pages 319–342. IGI Global, 2012.
16. J. S. Cuadrado, J. de Lara, and E. Guerra. Bottom-up meta-modelling: An interactive approach. In *MoDELS*, volume 7590 of *LNCS*, pages 3–19. Springer, 2012.
17. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.
18. R. Davis. Magic paper: sketch-understanding research. *Computer*, 40(9):34–41, 2007.
19. L. Dawson. A social-creative-cognitive (scc) model for requirements engineering. In *ISD*, 2012.
20. J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS'10*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010.
21. J. de Lara and E. Guerra. From types to type requirements: Genericity for model-driven engineering. *Software and Systems Modeling*, 12(3):453–474, 2013.
22. D. K. Deeptimahanti and M. A. Babar. An automated tool for generating uml models from natural language requirements. In *ASE*, pages 680–682. IEEE Computer Society, 2009.
23. J. Dingel, Z. Diskin, and A. Zito. Understanding and improving uml package merge. *Software and System Modeling*, 7(4):443–467, 2008.
24. Z. Diskin. Mathematics of UML: Making the Odysseys of UML less dramatic. In H. Kilov and K. Baclawski, editors, *Practical Foundations of Business System Specifications*, pages 145–178. Springer Netherlands, 2003.
25. Z. Diskin, B. Kadish, F. Piessens, and M. Johnson. Universal arrow foundations for visual modeling. In *Diagrams*, volume 1889 of *LNCS*, pages 345–360. Springer, 2000.
26. A. Dyck, A. Ganser, and H. Lichter. Model recommenders for command-enabled editors. In *MDEBE'2013*. CEUR, 2013.
27. J. Edwards. Example centric programming. *SIGPLAN Not.*, 39(12), Dec. 2004.
28. A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011.
29. M. Elaasar, L. C. Briand, and Y. Labiche. Domain-specific model verification with QVT. In *ECMFA*, volume 6698 of *LNCS*, pages 282–298. Springer, 2011. See also `https://sites.google.com/site/metamodelingantipatterns`.
30. M. Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.
31. S. Freeman and N. Pryce. *Growing Object-Oriented Software, Guided by Tests.* Addison-Wesley Professional, 1st edition, 2009.
32. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
33. R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, 1 edition, 2009. See also `http://www.eclipse.org/modeling/gmp/`.
34. J. L. C. Izquierdo and J. Cabot. Enabling the collaborative definition of DSMLs. In *CAiSE*, volume 7908 of *LNCS*, pages 272–287. Springer, 2013.
35. J. L. C. Izquierdo, J. Cabot, J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara. Engaging end-users in the collaborative development of domain-specific modelling languages. In *CDVE*, volume 8091 of *LNCS*, pages 101–110. Springer, 2013.
36. F. Javed, M. Mernik, J. Gray, and B. R. Bryant. MARS: A metamodel recovery system using grammar inference. *Inf. & Sof. Technology*, 50(9-10):948–968, 2008.
37. G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schneider, and S. Völkel. Design guidelines for domain specific languages. In *DSM'09*, pages 7–13, 2009.
38. S. Kelly and R. Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.
39. D. S. Kolovos, L. M. Rose, S. bin Abid, R. F. Paige, F. A. C. Polack, and G. Botterweck. Taming EMF and GMF using model transformation. In *MODELS'10*, volume 6394 of *LNCS*, pages 211–225. Springer, 2010.
40. M. Liquiere and J. Sallantin. Structural machine learning with galois lattice and graphs. In *ICML'98*, pages 305–313. Morgan Kaufmann, 1998.
41. S. Maoz, J. O. Ringert, and B. Rumpe. Modal object diagrams. In *ECOOP*, volume 6813 of *LNCS*, pages 281–305. Springer, 2011.
42. T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
43. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
44. Metamodel refactorings. `http://www.metamodelrefactoring.org/`.

45. O. Nierstrasz. Ten things I hate about object-oriented programming. *Journal of Object Technology*, 9(5), 2010.
46. OMG. UML 2.4.1 specification. `http://www.omg.org/spec/UML/2.4.1/`.
47. R. F. Paige, P. J. Brooke, and J. S. Ostroff. Specification-driven development of an executable metamodel in Eiffel. In *WISME'04*, 2004.
48. R. Perera. First-order interactive programming. In *PADL'10*, volume 5937 of *LNCS*, pages 186–200. Springer, 2010.
49. D. A. Sadilek and S. Weißleder. Towards automated testing of abstract syntax specifications of domain-specific modeling languages. volume 324 of *CEUR Workshop Proceedings*, pages 21–29. CEUR-WS.org, 2008.
50. C. Schäfer, T. Kuhn, and M. Trapp. A pattern-based approach to DSL development. In *DSM'11*, pages 39–46, 2011.
51. B. Shneiderman and C. Plaisant. *Designing the User Interface - Strategies for Effective Human-Computer Interaction (5. ed.)*. Addison-Wesley, 2010.
52. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley Professional, 2008.
53. M. Voelter. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. CreateSpace, 2013.
54. L. Wenyin, W. Zhang, and L. Yan. An interactive example-driven approach to graphics recognition in engineering drawings. *IJDAR*, 9(1):13–29, 2007.
55. R. Wirfs-Brock, L. R. Wiener, and B. Wilkerson. *Designing object-oriented software*. Prentice Hall, 1990.
56. D. Wüest and M. Glinz. Flexible sketch-based requirements modeling. In *REFSQ*, volume 6606 of *LNCS*, pages 100–105. Springer, 2011.