Bottom-up Meta-Modelling: an Interactive Approach

Jesús Sánchez Cuadrado[1], Juan de Lara[1], and Esther Guerra[1]

Universidad Autónoma de Madrid (Spain)
{Jesus.Sanchez.Cuadrado, Juan.deLara, Esther.Guerra}@uam.es

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. These should be tailored according to their future usage and specific implementation platform, which demands knowledge available only to engineers with great expertise in 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 model fragments by domain experts, with the possibility of using informal drawing tools like *Dia*. These fragments can be annotated with hints about the *intention* or *needs* for certain elements. A meta-model is 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.

Keywords: Meta-Modelling, Domain-Specific Modelling Languages, Interactive Meta-Modelling, Meta-Model Design Exploration

1 Introduction

Model-Driven Engineering (MDE) makes heavy use of models during the development process. Models are usually defined 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 is essential to obtain effective, useful meta-models [13].

The usual process of meta-model construction requires first building (a part of) the meta-model which only then can be used to build models. Even though software engineers are used to this process, it may be counter-intuitive to non-meta-modelling experts, which 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*

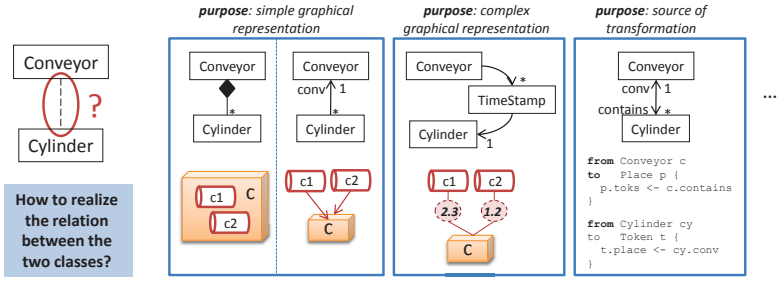


Fig. 1. Different meta-model realizations depending on its future usage

our minds" [19]. In this way, domain experts and final users of MDE tools are used to working with models, but not with meta-models. Asking them to build a meta-model *before* drafting example models is often too demanding. In general, an early exploratory phase of model construction, to understand the main concepts of the language, is recommended for DSML engineering [4, 13].

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 or Eugenia [14]), or the source or target of a transformation. As illustrated in Fig. 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 [24], 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* one 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 using a compact textual notation suitable for engineers which allows annotating the *intention* of the different modelling elements. From these fragments, a meta-model is automatically induced, which can be refactored if needed. Finally, the resulting

[1] `http://www.eclipse.org/Xtext/`
[2] `http://projects.gnome.org/dia/`

meta-model is compiled into a given technology (e.g., EMF or MetaDepth [8]), optimized for a particular *purpose* (visual or textual language, transformation) and a particular tool (e.g., xText or GMF).

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), and compilation of the induced meta-model for different purposes and platforms (Section 5). Next, Section 6 presents tool support. Finally, Section 7 compares with related research and Section 8 ends with the conclusions.

2 Bottom-up Meta-Modelling

Interactive development [21] 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 to explore 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 design from the beginning.

Inspired by interactive 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 [1]. 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. Afterwards, the induced meta-model can be reviewed by a meta-modelling expert who can refactor some parts if needed.
- *Interactive.* Creating a meta-model 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 [13]. The performed changes may require the detection of broken test models and their manual update. In our proposal, 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

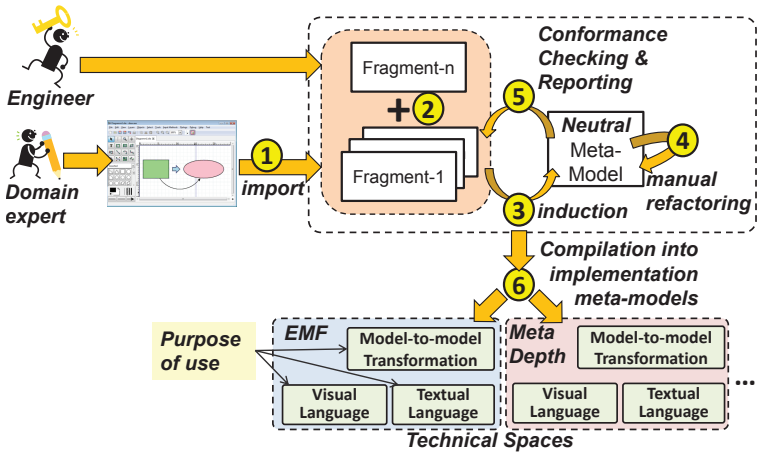


Fig. 2. Working scheme of bottom-up meta-modelling

exploration of design options, we should let the developer annotate his example models with hints about his intention, which are then translated into some meta-model design decision. If two models 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.

– *Implementation-agnostic.* The platform used to implement a meta-model may enforce certain meta-modelling decisions (e.g., the use of compositions vs references). This knowledge is sometimes not available to meta-modelling experts, but only to experts of the 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 process to build meta-models that is summarised in Fig. 2. First, a domain expert creates one or more example models using some tool with sketching facilities, such as Visio, PowerPoint or Dia. These examples are transformed into untyped model fragments made of elements and relations (step 1). An engineer can manipulate these fragments and define new ones, and also annotate his particular insight of certain elements in the fragments (step 2). A meta-model is 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 from a catalogue on the induced meta-model (step 4). In both cases, 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). Finally, in step 6, the user selects a platform and purpose of use, and

the neutral meta-model (and the fragments) is compiled into an implementation one, following the specific idioms of the target technical space. The compilation rules are customizable, and new compilations can be defined.

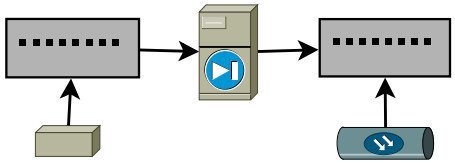
Realizing this approach poses several challenges. First of all, both engineers and non-technical experts need to develop model fragments. Engineers must be provided with a comprehensive set of annotations to specify design intentions. For non-technical experts, fragments are defined by sketches that 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, as well as a set of meta-model refactorings to enable the resolution of conflicts. Finally, we compile meta-models for specific platforms and uses, which requires studying the requirements of the considered platforms. These issues are discussed in Sections 3, 4 and 5.

3 Definition of Model Fragments

In our approach, users provide model fragments –examples of concrete situations– from which a meta-model is induced. Model fragments can be specified by a domain expert, typically using a drawing tool, or by an engineer, using a more concise syntax that can include annotations to guide the induction process.

As a running example, suppose we need to design a DSML to model simple factories and assist domain experts in modelling networks of machines. The machines receive and produce parts to conveyors, which can themselves be interconnected. Factories can include generators of two kinds of parts: dowels and cylinders. The left part of Fig. 3 shows a model fragment with a particular network as it would be sketched by a domain expert. It includes a machine connected to input and output conveyors, each transporting a different type of part. The right part of the figure shows the same fragment using the textual syntax that the engineer would use. Actually, we have an importer from *Dia* drawings that translates the sketches into textual fragments.

The textual syntax allows the engineer to enrich the fragments with *domain* and *design* annotations to guide the meta-modelling induction process. Domain annotations assign a meaning or feature to certain aspects of the fragment elements. For instance, the annotation *@container* attached to `Conveyor` indicates that, conceptually, conveyors are containers of items while these are being transported (see line 2 in Fig. 3). 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. Another example of domain annotation is *@global*, regarding the shareability of elements between different models. For example, a global clock may be used to synchronise all simulation models of a system. If an element is not tagged as *@global*, it is assumed to be local, i.e., accessible in the scope of the current model only.



```
1  fragment "conveyorSequence" {
2    @container
3    cin: Conveyor {
4      rel asm = a;
5    }
6    cout: Conveyor {}
7    a: Assembler {
8      rel outs = cout;
9    }
10   p1: Dowel {
11     rel conv = cin;
12   }
13   p2: Cylinder {
14     @general
15     rel conv = cout;
16   }
17 }
```

Fig. 3. Model fragment definition: Graphical concrete syntax used by the domain expert (left), and compact textual syntax used by the engineer (right)

These annotations may be translated later into meta-modelling design decisions or used to guide the meta-model compilation process.

On their side, *design* annotations refer to meta-modelling decisions that should be reflected in the meta-model generated from the fragments. These decisions can also be given later by refactoring the induced meta-model, but the engineer is given the possibility to define them in advance using annotations. For instance, 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 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. For example, the annotation in Fig. 3 (line 14) will cause the creation of a parent class for dowels and cylinders, defining the common reference `conv`. Other examples of design annotations are *@external*, to indicate cross-references, or *@partial*, to indicate that a class is only partially defined and should be completed with others through merging or inheritance.

Altogether, annotations are a means to record an insight of the engineer at a given point in the running session, and will be used at some point in the future to guide the meta-model induction process.

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. Next, we describe our meta-model induction algorithm, how meta-model refactorings are applied, and the strategy for conflict resolution.

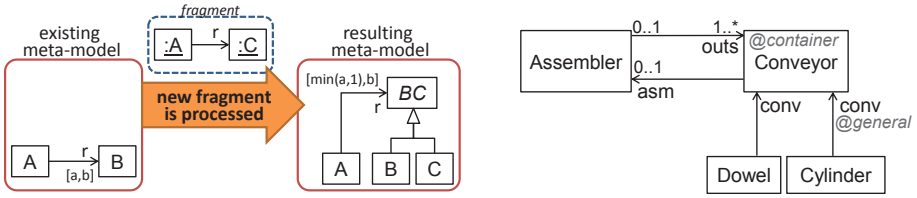


Fig. 4. Processing a reference with different target type in the meta-model and a fragment (left). Meta-model induced from the fragment in Fig. 3 (right)

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, 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 relation stemming from an object, a relation type is created in its meta-class, if it does not exist. The minimum cardinality of relations is set to 0 by default, or to 1 if all objects of the same kind define the reference. The maximum cardinality can be set to 1 or unbounded. We take the convention of mapping plural reference names to multivalued references, and singular to monovalued ones.

If two relations with the same name point to objects of different type, our algorithm creates an abstract superclass as target of the relation type, with a subclass for the type of each target object. This situation is illustrated to the left of Fig. 4, where the new abstract class `BC` is created as parent of both `B` and `C`. In this example, `BC` would not be created if the `B` meta-class is abstract and the `C` object defines features that are compatible with those in `B`. The minimum cardinality of the relation 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 minimum cardinality (value `a`) may be zero. The maximum cardinality `b` of the relation is kept. As we will explain in Section 4.3, any automatic design decision made by the algorithm is reported to the user, so that he can change it.

As an example, Fig. 4 shows to the right the meta-model induced from the fragment in Fig. 3. According to the heuristic, the `conv` and `asm` relations (*singular*) were assigned upper bound 1, while `outs` (*plural*) received upper bound *. For the lower bound, it is 0 in `asm` because the fragment contains conveyors not connected to assemblers, but it is 1 in `outs` because all assemblers are connected to some conveyor in the fragment. Additionally, the *@general* and *@container* annotations were copied from the fragment to the meta-model.

4.2 Refactoring of meta-models

The annotations transferred from the fragments to the meta-model may trigger refactorings in it to reflect the annotated intentions. For example, the left of

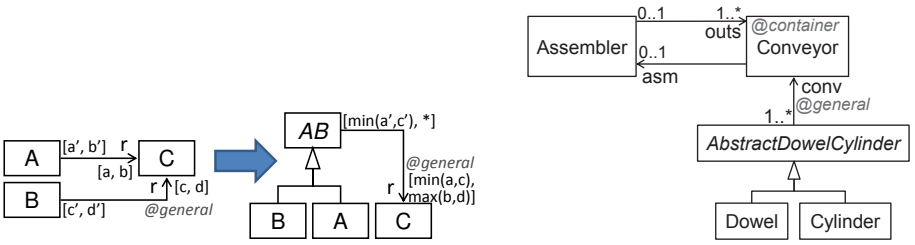


Fig. 5. Scheme of the refactoring triggered by *@general* (left). Result of the refactoring on the meta-model to the right of Fig. 4 (right)

Fig. 5 shows the refactoring triggered by the *@general* annotation, which is similar to the *pull-up* refactoring [10]. It pulls up the annotated attribute or relation as general as possible in an inheritance hierarchy. If the annotated attribute or relation 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 [10] is applied). The target of the pulled relation receives as lower bound the minimum of the lower bounds, and as upper bound the maximum of the upper bounds. The situation is similar for the source, but in this case the upper bound is generalised to * as the pulled relation must merge those from `A` and `B` (i.e., we take * as the upper bound instead of the sum of the upper bounds `b'+d'`).

The right of Fig. 5 shows the result of executing this refactoring to the meta-model in Fig. 4, due to the *@general* annotation in reference `conv`. A new abstract class `AbstractDowelCylinder` is created as parent of both `Dowel` and `Cylinder`, acting as target of the reference.

4.3 Interactivity and exploration by supervising decisions

Our induction process and the triggered refactorings are automated mechanisms. If there are several design alternatives available, 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. The user may apply refactorings manually, through a catalogue of refactorings provided as part of the environment, if his level of expertise allows him to decide how to change the meta-model.

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 [5], 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 three kinds of open issues: *conflict*, *automatic* and *suggestion*, which are briefly explained next.

Conflict. The definition of new fragments may imply the update of the meta-model. For example, the cardinality of existing relations may need to be changed, or new classes may need to be created. If a fragment contains contradictory information, e.g., if the same attribute is assigned incompatible types in different objects, then a conflict arises. For instance, there is a conflict if a conveyor defines an attribute `attr id=‘‘c1’’`, and another conveyor defines `attr id=2`. 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., the conveyor with `id=2` would be automatically changed to `id=‘‘2’’`), while a change from String to Integer is breaking and unresolvable, and requires the intervention of the user. Our algorithm chooses by default an alternative that is non-breaking or, at least, resolvable.

Automatic. These are decisions automatically taken by the induction algorithm when several alternatives exist. For instance, the name of the superclass automatically introduced for `Dowel` and `Cylinder` is built by concatenating the subclasses' names prefixed by "Abstract" (i.e., `AbstractDowelCylinder`). The user is notified about this design decision, and is offered the possibility of changing the superclass' name.

Suggestion. Some meta-model improvements may be possible, like the application of guidelines or meta-model design patterns. These are provided as suggestions which, if accepted, will trigger a certain meta-model refactoring. As an example, if a reference is multivalued but its name is singular, our engine will suggest the user to give it a plural name. So far, we support simple suggestions, but our aim is to define and implement a catalogue of meta-modelling good practices that help non-expert users improve the quality of their meta-models.

5 Meta-Model 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 (e.g., if the models will be edited with the default tree editor), making heavy use of composition associations. If we aim at creating a

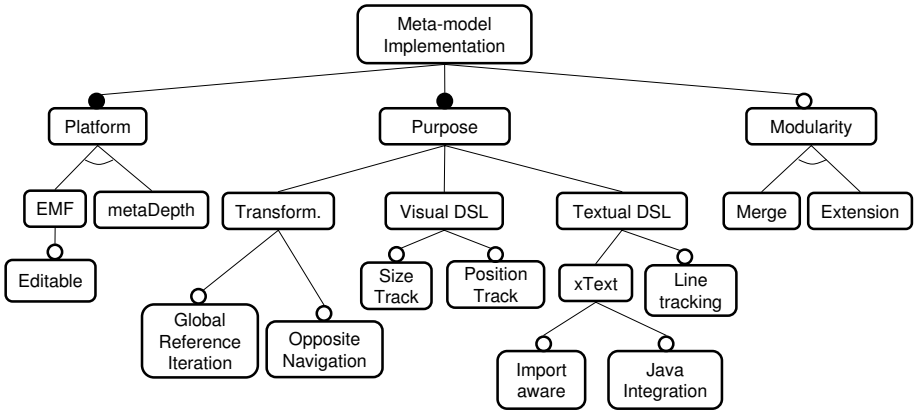


Fig. 6. Feature model for meta-model compilation (excerpt)

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 such a neutral, conceptual meta-model into implementation ones for specific platforms and purposes.

Fig. 6 shows a feature model that gathers some compilation variants from our 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, which can help to fine-tune the compilation. The implementation of the modularity mechanism for meta-models is also subject of two variants: package merge or cross-references between meta-models.

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

- **EMF platform**. This compilation produces an *ecore* meta-model. Optionally, by setting the *Editable* flag equal to *true*, the compilation generates a *root* class and composition associations to allow any class to be reachable from the root class via composition associations. Each reference from a class annotated with *@container* to a class annotated as *@containee* is converted to a composition. Finally, EMF uses references instead of full-fledged associations, and references can only have cardinalities at the target end. For this reason, this compilation generates two opposite references for those relations in the neutral meta-model with cardinality different from * in the source.
- **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. In contrast to EMF, MetaDepth does not support composition, therefore the compilation generates OCL constraints for those references between *@container* and *@containee* objects.
- **Transformation profile**. In this profile, we can configure two aspects to optimize navigation expressions. By selecting *Opposite Navigation* all 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. In this case, an intermediate class is generated to permit the iteration. If MetaDepth is selected as target platform, this option generates an *Edge* instead.
- **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, Fig. 7 shows to the left the neutral meta-model obtained by the induction process. This meta-model was obtained from the fragment in Fig. 3 and two additional fragments: one specifying a `Generator` connected to a `Conveyor`, which produced the `Machine` abstract class, and another one connecting two `Conveyors`, which generated the `nexts` relation. The `AbstractDowelCylinder` class was renamed to `Part` through a *renaming* refactoring. Since none of the new fragments contained `Part`s, the lower bound of the source of `conv` was set to zero by the algorithm. Finally, an additional *@containee* annotation was manually added to `Part`. The figure also shows the compilation of this meta-model into EMF using the transformation profile. For the EMF platform, we chose the generation of a root class, and the reference between the *@container* and *@containee* was compiled into a composition. Additionally, the *transformation* profile makes each reference bidirectional. A wizard asks the user any information needed to complete the compilation, like the name of the root class and association ends. Moreover, all model fragments are compiled using the same options, so that a set of testing models becomes available for free.

6 Tool Support

Realizing our approach requires specialized, integrated tool support that has to go beyond the dominant style of meta-modelling nowadays: top-down and based on a batch processing style. To this end, we have implemented a tool for Eclipse that gives interactivity to our approach[3]. Next, we describe the elements of our tool by going through an interaction example that is shown in Fig. 8.

[3] Available at `http://sanchezcuadrado.es/projects/interactive-metamodeling`

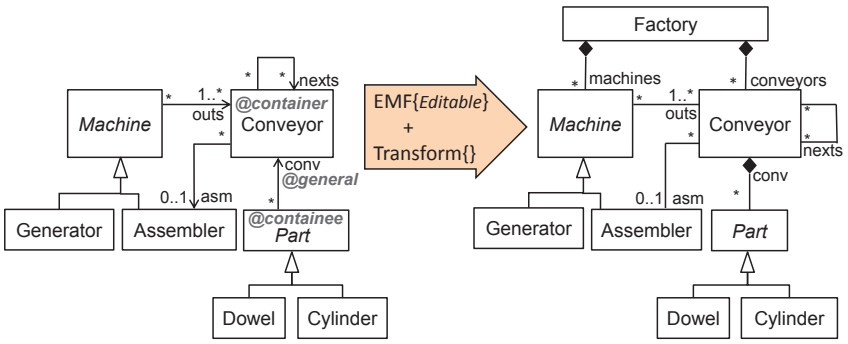


Fig. 7. Compiling to EMF for transformation

1. Sketching fragments. If a domain expert is involved in the meta-modelling process, he may first sketch fragments that represent scenarios in the domain. We have implemented an import facility that takes a diagram sketched with the Dia tool and generates a model fragment ready to be evaluated. This facility implements simple heuristics to determine sensible names for objects and references, and takes advantage of the visual containment relationships between elements to generate equivalent annotations.

A drawing tool such as Dia (and others like Visio) offers a wide variety of symbols, organized in categories, that can be used to sketch fragments. However, we do not expect that a non-technical user respects the semantic meaning of the symbols (in the figure, the symbol to depict a conveyor is actually a representation of a network hub). Instead, a symbol is used if the pictogram resembles what the user wants to convey. Nonetheless, it is important that each meta-model element is assigned a meaningful name within the domain. For this purpose, symbols can be attached a legend with their name.

2. Editing fragments. Engineers can create fragments by using the textual notation introduced in Section 3. Actually, the sketched fragments are translated into this second textual notation for their subsequent edition and manipulation, e.g., to add annotations or to refine the name of the types. In the example, the engineer has added the `@container` and `@general` annotations, as well as a new attribute `id` to `Conveyors`. Textual fragments are edited by using a text editor (built with xText) with syntax highlighting, error reporting, templates and autocompletion. A key binding and the "Update metamodel" menu option permits processing the current fragment.

3. Visualizing the meta-model. Processing a fragment induces a new version of the meta-model. However, there is no editor to modify the meta-model. Instead, the meta-model is visualized so that the user can check its state and evolve it in three ways: processing new fragments, applying manual refactorings or addressing open issues. Implementation-wise, the current version of our tool uses the Zest framework to render and layout meta-models.

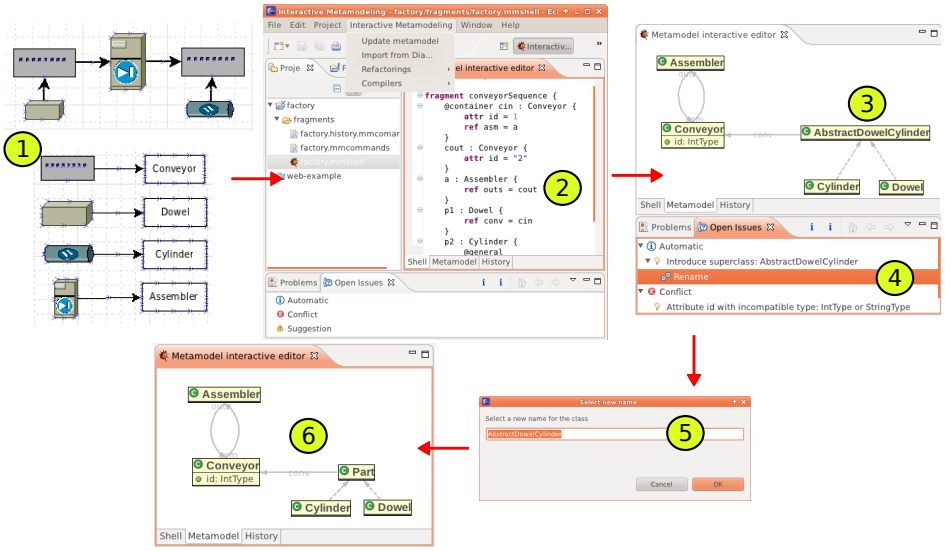


Fig. 8. Example of interaction with the tool

4. Addressing open issues. The interactive and exploratory nature of our approach is realised through the *open issues view*. This view gives the user information about conflicts as well as suggestions of possible refactorings. Selecting an issue will show possible fixes that in turn will launch a refactoring. In the figure, two issues are reported: (1) the introduction of a new superclass, and (2) a conflict related to incompatible types for the `id` attribute. For the first issue, our system proposes a rename action (if needed). Additionally, every command and updated fragment is recorded and can be queried in the history tab. Sessions are persistent, that is, the state of the session is stored after each evaluation so that it can be interrupted and resumed later.

5. Applying a refactoring. Selecting a proposal from the open issues view will raise a refactoring. The refactoring may require the user to provide some information, as is the case of the figure, where the name of the class is asked (in this case `Part` is given). Afterwards, the visualization of the meta-model is automatically updated (step 6).

These steps are performed iteratively until all concepts of the domain are represented in the meta-model. At this point, the meta-model can be compiled for some particular purpose, as explained in Section 5. The user then selects the purpose (for instance, creating a textual DSML with xText), and the environment automatically selects dependent features (e.g., xText implies EMF) and shows a *wizard* that asks the user for optional features. Finally, the meta-model is generated in the selected format, for instance Ecore.

7 Related Work

There are some works dealing with the inference of meta-models from models. The MARS system [12] 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 [15]. 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 [7], 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 [20], 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 [22], 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 [18]. We provide support for many of them. Finally, the idea of testing meta-models by creating test cases is orthogonal to our approach, and could be integrated in our environment.

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 [4], 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 [3]. 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 and different compilations into *implementation* meta-models, guided through annotations and selection of configurations.

In our approach, newly introduced fragments may raise conflicts if the fragments contain contradictory information. Some application domains where the resolution of conflicts has been extensively studied are model merging [17], change propagation in software systems [9] and distributed development [6]. It is up to future work to identify how the conflicts that may arise when evolving a meta-model relate to these previous works.

Another line of related work concerns the expressiveness of model fragments. While one could simply use object diagrams, in [16], 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 different as we use fragments to automatically induce a meta-model. While we consider negative fragments, they are not yet taken into account by the induction algorithm.

A way to simplify and make the development of meta-models systematic is through design patterns. In [2], some design patterns for meta-models are proposed, while in [23], the requirements for meta-models are represented as use case diagrams and the meta-models are evolved by applying patterns. We plan to integrate patterns in our approach to guide the induction phase and refactor meta-models towards patterns. Integrating end-users in the meta-model construction process has also been regarded as a means to improve the quality of the resulting meta-model. In [11], the authors propose a collaborative approach to meta-model construction which involves both domain and technical experts. The approach is supported by a DSL to represent the collaborations among stakeholders (change proposals, solution proposals or comments) while the meta-model is being developed.

Finally, our meta-model refactorings and subsequent propagation to the model fragments can be seen as a simplified scenario of meta-model/model evolution [5].

8 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. A specialized textual notation is also provided for advanced users, who can annotate the fragments to guide the automatic induction of the meta-model. The process is iterative, as fragments are added incrementally, causing updates in the meta-model, which can be refactored in the process. Finally, the meta-model can be 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. 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. 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.

Acknowledgements. This work was funded by the Spanish Ministry of Economy and Competitivity (project “Go Lite” TIN2011-24139) and the R&D programme of the Madrid Region (project “e-Madrid” S2009/TIC-1650).

References

1. C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity*, volume 1. The MIT Press, 2000.
2. H. Cho and J. Gray. Design patterns for metamodels. In *DSM'11*, 2011.
3. H. Cho, J. Gray, and E. Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *MiSE'12*, 2012.
4. 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.
5. A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating co-evolution in model-driven engineering. In *EDOC'08*, pages 222–231, 2008.
6. A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Managing model conflicts in distributed development. In *MODELS'08*, volume 5301 of *LNCS*, pages 311–325. Springer, 2008.
7. A. Cicchetti, D. D. 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.
8. J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS'10*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010.
9. A. Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE TSE*, 37(2):188–204, 2011.
10. M. Fowler. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.
11. J. L. C. Izquierdo and J. Cabot. Community-driven language development. In *MiSE'12*, 2012.
12. 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.
13. 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.
14. 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.
15. M. Liquiere and J. Sallantin. Structural machine learning with galois lattice and graphs. In *ICML'98*, pages 305–313. Morgan Kaufmann, 1998.
16. S. Maoz, J. O. Ringert, and B. Rumpe. Modal object diagrams. In *ECOOP*, volume 6813 of *LNCS*, pages 281–305. Springer, 2011.
17. T. Mens. A state-of-the-art survey on software merging. *IEEE TSE*, 28(5):449–462, 2002.
18. Metamodel refactorings. `http://www.metamodelrefactoring.org/`.
19. O. Nierstrasz. Ten things I hate about object-oriented programming. *Journal of Object Technology*, 9(5), 2010.
20. R. F. Paige, P. J. Brooke, and J. S. Ostroff. Specification-driven development of an executable metamodel in Eiffel. In *WISME'04*, 2004.
21. R. Perera. First-order interactive programming. In *PADL'10*, volume 5937 of *LNCS*, pages 186–200. Springer, 2010.
22. 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.

23. C. Schäfer, T. Kuhn, and M. Trapp. A pattern-based approach to DSL development. In *DSM'11*, pages 39–46, 2011.
24. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, 2008.