# Example-based generation of graphical modelling environments

Jesús J. López-Fernández, Antonio Garmendia, Esther Guerra, Juan de Lara

Universidad Autónoma de Madrid (Spain)

**Abstract.** Domain-Specific Languages (DSLs) present numerous benefits like powerful domain-specific primitives, an intuitive syntax for domain experts, and the possibility of advanced code generation for narrow domains. While a graphical syntax is sometimes desired for a DSL, constructing graphical modelling environments is a costly and highly technical task. This relegates domain experts to play a passive role in their development and hinders a wider adoption of graphical DSLs.
Targeting a simpler DSL construction process, we propose an example-based technique for the automatic generation of modelling environments for graphical DSLs. This way, starting from examples of the DSL likely provided by domain experts using drawing tools like yED, our system is able to synthesize a graphical modelling environment that mimics the syntax of the provided examples. This includes a meta-model for the abstract syntax of the DSL, and a graphical concrete syntax supporting spatial relationships like containment or attachment. The system is implemented as an Eclipse plugin, and we demonstrate its usage on a running example in the home networking domain.

**Keywords:** Domain-Specific Modelling Languages, Graphical Modelling Environments, Example-Based Meta-Modelling, Flexible Modelling.

## 1 Introduction

Model-Driven Engineering (MDE) is founded on the use of models to describe the systems to be built. Often, these models are defined using Domain-Specific Languages (DSLs) tailored to a particular field [7]. Hence, the need to create DSLs and their associated modelling environments is recurring in MDE projects.

The concrete syntax of a DSL may be graphical or textual, though in this paper we focus on graphical DSLs [10]. Many tools have emerged along the years to build environments for graphical DSLs [3, 5, 6, 10, 11, 17]. However, building such environments still remains a technical, complex and time-consuming task. For example, building a graphical editor with Graphiti [5] requires manual programming based on a large Java API. In the case of GMF [6] and Sirius [17], it is necessary to describe the different aspects of the editor by building one or more models. These models may become very detailed, large and hard to build and maintain for non-experts – especially for DSLs beyond toy examples – and frequently they must be constructed using unhandy tree-based editors.

Apart from the technical difficulties, a salient issue with most graphical language workbenches is the need to construct a meta-model upfront, and to describe the features of the concrete syntax and the modelling environment using a technical language or notation. This hinders the active participation of domain experts in the DSL construction process, who might find more familiar working with examples than with meta-models [1, 12] and might lack the technical knowledge to define complex environment specifications. However, the active involvement of domain experts is crucial for the success of the DSL to be built [9].

To avoid these difficulties, we propose a novel technique for the automatic generation of graphical modelling environments starting from examples of the DSL. Hence, instead of building a meta-model first and describing its concrete syntax at the meta-model level, our proposal is to collect examples built by domain experts using drawing tools like Powerpoint, Dia or yED. Our framework processes the provided examples to induce a meta-model by using the techniques presented in [12], and it also extracts a description of the graphical concrete syntax that includes graphical forms for classes (*svg* files), edge styles, and spatial relations like containment or attachment. This information is used to synthesize a graphical modelling environment that mimics the graphical syntax used in the examples, but in addition, it enforces the well-formedness rules of the DSL and enables the creation of models (in contrast to drawings) that can be manipulated using MDE technology (e.g., transformations and code generators). As a result, a graphical DSL environment is generated with no need to code or create complex technical specifications. Our proposal is backed by a working prototype, available as an Eclipse plugin at `http://miso.es/tools/metaBUP.html`.

**Paper organization**. Section 2 presents an overview of our approach and a running example. Section 3 introduces example-based meta-modelling. Section 4 shows our approach to extract concrete syntax information from graphical examples. Section 5 describes the synthesis of graphical modelling environments from the extracted information. Section 6 presents tool support. Finally, Section 7 discusses related research and Section 8 concludes the paper.

## 2 Overview and running example

Fig. 1 outlines our process for the example-based generation of graphical modelling environments. It involves two roles: the *Domain Expert*, who provides graphical examples and ultimately validates the generated environment, and the *Modelling Expert*, who monitors the meta-model induction process from which the desired DSL environment is derived.

The core part of our process, gray-shaded in Fig. 1, is iterative. Here, the domain expert provides input examples made with tools like yED, portraying how models should look like (label 1). These examples may represent complete models, or they may focus on a particular aspect of interest and therefore be partial, in which case we call them *fragments*. Then, the examples are automatically parsed into models, which are more amenable to manipulation (label 2). The parsed models are represented textually, making explicit the existing

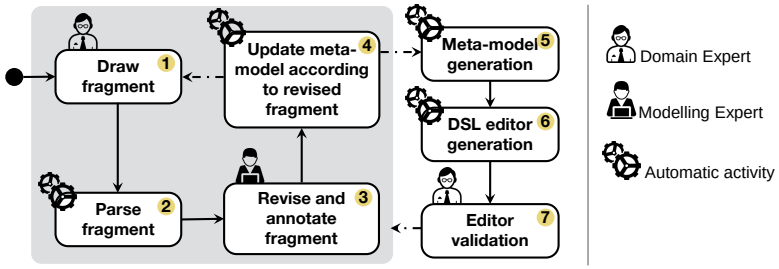


Fig. 1: Bottom-up graphical DSL development process.

objects, attributes and relations in the examples, annotated with information regarding their graphical rendering (e.g., spatial relationships between objects or line styles). The modelling expert can edit this textual representation (label 3) to set more appropriate names to the derived relations, or to trigger refactorings in the meta-model induction process which takes place next (label 4). Thus, an iteration step finishes when the meta-model under construction is evolved to accept the revised fragment.

After processing all provided examples, the modelling expert can export the induced meta-model to a suitable format (Ecore in our current implementation, label 5), and invoke our editor generator to obtain a fully operating editor mimicking the concrete syntax of the examples (label 6). Moreover, the examples are migrated into models and can be edited and visualized in the generated editor. The domain expert can validate the editor (label 7), perhaps based on the converted examples, and if necessary, he can refine the DSL by providing further examples and re-generating the editor.

### 2.1 Running example

As a running example, we develop a DSL in the home networking domain. In this DSL, we would like to represent the contracts that internet service providers (ISPs) hold with customers, the possible configurations of home networks, and their connection with the ISP infrastructure. Customer homes are connected via cable modems to the ISP network. Typically, each home has a (normally Wi-Fi-enabled) router to which the landline phone is connected, and with a number of Ethernet cable ports. Wi-Fi networks are password protected and work in a frequency range. Moreover, each home may have both cabled (e.g., PCs, printers or laptops) and wireless devices (e.g., smartphones, tablets or laptops).

Using our approach, domain experts provide example fragments that illustrate interesting network configurations and depict the desired graphical representation for them. As an example, Fig. 2 shows one fragment built with yED[1], representing the connection between some customer homes and the ISP through cable modems. The elements in the drawing define some properties, like the ipBase of cable modems, the name of the home owner, the tier and location of the

[1] `https://www.yworks.com/products/yed`

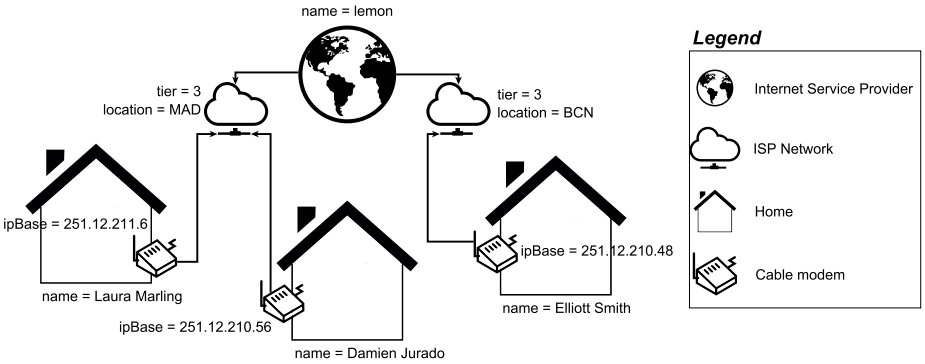


Fig. 2: Fragment showing a connection between customer homes and an ISP.

ISP network, and the name of the ISP. The legend to the right assigns a name to every picture used in the drawing.

## 3 Example-based meta-modelling

In [12], we introduced a bottom-up meta-modelling technique that enables the automatic induction of a meta-model starting from sketches[2], built using drawing tools. In order to facilitate the meta-model induction process, sketches are complemented by a legend that assigns a name to each different symbol in the drawing, as shown in Fig. 2. Such names are used as identifiers for the induced meta-model classes.

The meta-model induction process starts by parsing the provided fragment into a textual internal representation that is easier to manipulate by the modelling expert. The fragment, once revised by the modelling expert, is fed into our system. This may produce an update of the current version of the meta-model so that it "accepts" the provided fragment. For example, if a fragment contains objects of an unknown type, this type is incorporated into the meta-model. Similarly, if an object has new features not present in its type, then its meta-class is extended with these new features. Fragments have an *open-world* semantics: they only convey the relevant information for the scenario, and may omit additional information that will be given in further sketches. As explained in Section 2, examples are a special kind of fragments used to represent complete models, and they have a *closed-world* semantics.

For instance, Listing 1 shows the textual model obtained from parsing the fragment in Fig. 2. Every object (e.g., h1 in line 2) receives a type as indicated in the legend (e.g., Home), and may contain slots (e.g., name in line 3) and links (e.g., modem in line 6) according to the original fragment.

[2] We call these examples *sketches* to distinguish them from models conformant to a meta-model, though they are not hand-drawn but made with diagramming tools.

```
1  fragment fragment1 {
2    h1 : Home {
3      attr name = "Elliott Smith"
4      @overlapping
5      @composition
6      ref modem = cm3
7    }
8    isp1 : InternetServiceProvider {
9      attr name = "lemon"
10     ref infrastructure = ispn1, ispn2
11   }
12   h2 : Home {
13     attr name = "Damien Jurado"
14     @overlapping
15     @composition
16     ref modem = cm2
17   }
18   h3 : Home {
19     attr name = "Laura Marling"
20     @overlapping
21     @composition
22     ref modem = cm1
23   }
24   cm1 : CableModem {
25     attr ipBase = "251.12.211.6"
26     ref isp = ispn1
27   }
28   cm2 : CableModem {
29     attr ipBase = "251.12.210.56"
30     ref isp = ispn1
31   }
32   cm3 : CableModem {
33     attr ipBase = "251.12.210.48"
34     ref isp = ispn2
35   }
36   ispn1 : ISPNetwork {
37     attr tier = 3
38     attr location = "MAD"
39   }
40   ispn2 : ISPNetwork {
41     attr tier = 3
42     attr location = "BCN"
43   }
44 }
```

Listing 1: Textual representation of the fragment in Fig. 2

Fig. 3 shows the meta-model induced from this fragment. As this is the first fragment, the meta-model was initially empty, and so four new classes are added, each containing the necessary attributes for the slots in the class' objects. We use simple heuristics to type primitive attributes, like setting the type to int when all slots within a fragment are compatible with that type (e.g., tier in the example). If a subsequent fragment invalidates such an assumption, then the type will be changed to String. References are assigned cardinality * as soon as an object points to two or more objects using edges with the same style (e.g., infrastructure). We also detect spatial relations between objects, like overlapping and containment, in which case compositions are created in the meta-model. In the example, the system detects overlapping between each CableModem object and a Home object.

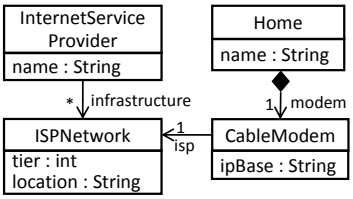


Fig. 3: Meta-model induced from the fragment in Listing 1.

Objects, slots and links in the textual fragment can be annotated manually by the modelling expert. Such annotations can provide design or domain information accounting for well-formedness constraints of the DSL (see [12]), or they can refer to concrete syntax details. In addition, some concrete syntax annotations are automatically produced by the fragment importer. In Listing 1, the importer added annotation @overlapping in lines 4, 14, and 20, to convey the fact that Home and CableModel objects overlap each other. We will detail the use of this kind of annotations in Section 4. In [12], we reported on another use of annotations, as a means to encode meta-model integrity constraints, like @composition in lines 5, 15, and 21. As we will see in Section 4, the @composition annotation was heuristically added due to the existence of overlapping.

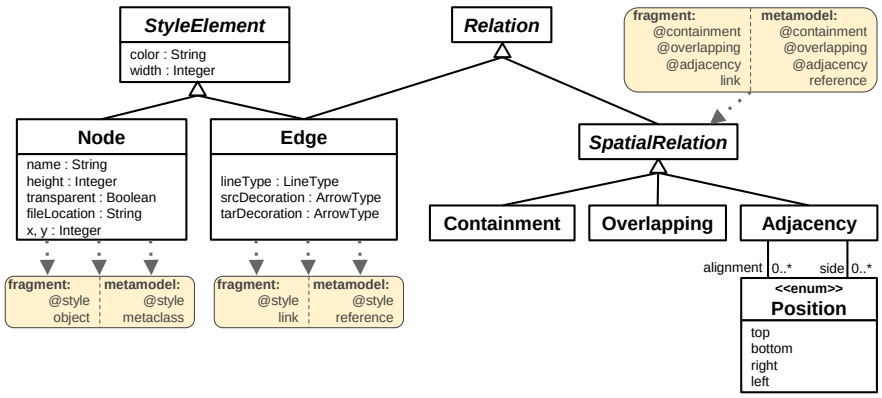


Fig. 4: Graphical properties inferable from sketches, and corresponding annotations.

The meta-model changes after each fragment is processed may trigger recommendations (refactorings). For example, if two classes have similarities (common attributes or references pointing to the same class) the system suggests applying the *extract superclass* refactoring, to factor out the common information [12].

Our technique is incremental, as new examples and fragments can be provided to make the meta-model evolve. Moreover, it fosters the active participation of domain experts in the meta-model construction process, as they can contribute with fragments (sketches) which are no longer passive documentation, but they are used to derive a meta-model. Up to now, our technique has been only able to derive the abstract syntax of the DSL [12]. In the following, we elaborate on the main contribution of this paper, which is the extension of our approach to derive a concrete syntax for the DSL (Section 4) and to synthesize a graphical modelling environment that emulates the syntax of the fragments (Section 5).

## 4 Example-based concrete syntax inference

We take advantage from the graphical information already encoded in sketches for both minimising the job of the modelling expert and deriving a concrete syntax close to the domain expert's conception.

Fig. 4 shows the graphical properties that we extract from sketches and use to derive the concrete syntax of the DSL. Some are explicit features from the icons in the drawing, like their colour or size. Other properties are implicit relationships concerning the relative position of icons, like overlapping or adjacency, and are derived automatically by studying the size and location of each icon. For adjacency, we check both the direction (e.g., two objects adjacent left-to-right) and if in addition they are aligned and how (e.g., at the bottom).

Graphical properties are encoded as annotations of the corresponding objects and links in the textual fragment. Then, these annotations are transferred to the appropriate meta-model classes and references when the fragment is processed. Fig. 4 shows the correspondence between the graphical properties and the elements they can annotate.

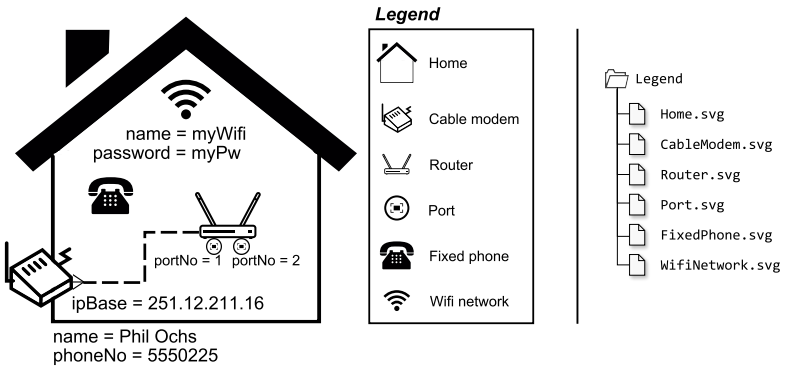


Fig. 5: Fragment with spatial features (left). Content of the *legend* folder (right).

Next, we explain how we extract and manipulate this graphical information.

### 4.1 Detection of icons and line styles

We retrieve each icon employed in the provided sketches, since this is the most relevant aspect of the appearance that the domain expert expects from the final DSL. Since the drawing tools we work with demand the definition and usage of palettes with all available icons, technically, we provide a directory where we store a copy of the files containing the icons as they are added to the palette. These files are employed both in the serialization of fragments and in the generation of the concrete syntax, and are named according to the icon they contain. For instance, Fig. 5 shows to the right the *Legend* folder that contains the *svg* files used to represent each domain object in the fragment to its left.

Additionally, we detect and classify the style of edges in sketches. This feature can be deactivated if the edge style is irrelevant for the domain. If active, we identify and record the colour, line width, style (e.g., dotted) and source and target decorations of edges. As an example, Fig. 5 contains an edge linking a Router and a Cable modem. When the fragment is imported, the link is annotated with the identified style (lines 26–28 in Listing 2).

Note that the name inferred for this link was not modem, but the one struck out (see lines 29–30 in Listing 2). Because we allow the modelling expert to edit the text fragments, he has replaced the inferred name with one closer to the domain. What is interesting about this operation is that, from this moment on, each time a link with the same style between a router and a cable modem is imported, it will be automatically named modem. If the modelling expert renames the feature in the future, he will be offered two options: either to replace the previous name modem with the new one, or creating a new reference in class Router which would coexist with the feature modem.

The annotations with the graphical information of links will be transferred to the corresponding meta-model references, and eventually, to the concrete syntax generator. On the contrary, meta-classes do not carry any graphical information with them, since we store their exact representation in the legend folder.

```
1  fragment fragment2 {
2    Home_1 : Home {
3      attr phoneNo = 5550225
4      attr name = "Phil Ochs"
5
6      @overlapping
7      @composition
8      ref modem = CableModem_1
9
10     @containment
11     @composition
12     ref electronicDevices = Router_1
13
14     @containment
15     @composition
16     ref phones = FixedPhone_1
17
18     @containment
19     @composition
20     ref wifiNetworks = WifiNetwork_1
21   }
22   Router_1 : Router {
23     @adjacency(side = bottom)
24     ref ports = Port_1, Port_2
25     @composition
26     @style ( color = "#000000", width = 3,
27         line = dashed, source = none,
28         target = crows-foot-many )
29     ref '00000_3_dashed_none_crows-foot-many'
30         modem = CableModem_1
31   }
32   FixedPhone_1 : FixedPhone { }
33   WifiNetwork_1 : WifiNetwork {
34     attr name = "myWifi"
35     attr password = "myPw"
36   }
37   Port_1 : Port { attr portNo = 2 }
38   Port_2 : Port { attr portNo = 1 }
39   CableModem_1 : CableModem {
40     attr ipBase = "251.12.211.16"
41   }
42 }
```

Listing 2: Textual representation of the fragment in Fig. 5

### 4.2 Detection of spatial relationships

Sometimes, spatial relationships between graphical objects have a meaning in the domain and need to be modelled. It is even likely that the domain expert is unaware of whether layout implies domain requirements. We automatically detect spatial relationships in sketches, and leave the modelling expert to keep or discard them by editing the textual fragments. We currently support three kinds of spatial relationships:

- *Containment*: a graphical object is within the bounds of another.
- *Adjacency*: two graphical objects are joined or very close. The maximum distance with which adjacency is to be considered is user-defined (*0* by default). Two optional properties are likewise detected: the side(s) from which objects are attached to each other, and alignment, a special type of adjacency.
- *Overlapping*: two graphical objects are superimposed (but not contained).

Detecting one of these relationships implies adding a reference to the metamodel. In the case of containment, the reference goes from the container to the containee. For adjacency and overlapping, we use this heuristic: if an object `o` overlaps (or is adjacent) to more than one object of the same kind, the reference stems from `o`'s class; otherwise, the reference stems from the class of the bigger object. The rationale is that, frequently, the different parts of bigger objects are represented as smaller affixed elements (e.g., a component with affixed ports).

The fragment in Fig. 5 illustrates all supported spatial relationships, which are automatically detected when the fragment is imported (see Listing 2). On one hand, the Home contains a Router, a Fixed Phone and a Wifi Network in the sketch; hence, in the textual representation, the Home object has three links annotated as @containment (lines 12, 16 and 20). The Home overlaps with a Cable Modem in the sketch, being the Home icon bigger; hence, the Home object is added a

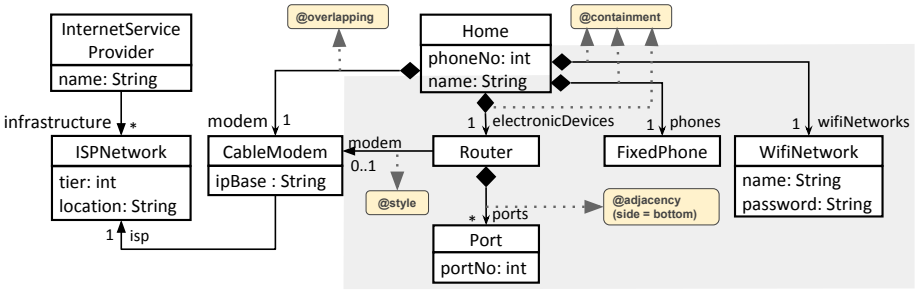


Fig. 6: Updated meta-model after processing the fragment of Listing 2.

link annotated as @overlapping (line 8). Finally, the Router has two adjacent Ports to the bottom side; since there are multiple ports, the Router is added a link annotated as @adjacency (line 24). The side parameter of this annotation could be removed in case the side of the adjacency is irrelevant to the domain.

In addition to creating explicit links for the detected spatial relationships, our importer heuristically adds @composition annotations to the created links (see lines 7, 11, 15, 19 and 25). This helps in organizing and realising only a *sufficient* set of spatial relationships. For example, both Ports are contained in the Home, but this relation is not made explicit because they are already adjacent to the Router, which is inside the Home. In this case, we use the @composition annotation of the abstract syntax to infer that they are indirectly contained in Home objects.

Fig. 6 shows the resulting meta-model after processing this second fragment, including the annotations for style properties and spatial relationships. The new features with respect to Fig. 3 appear gray-shaded.

## 5 Generation of graphical modelling environments

Our approach to synthesize the graphical editor proceeds in two steps: we first convert the information gathered from the sketches into a technology-neutral graphical representation, and then, this representation is translated into a technology-specific editor specification. We currently target Sirius [17], but other technologies like EuGENia [11] could be easily targeted as well. Fig. 7 outlines this process, where three transformations take place: one generates the meta-model with the abstract syntax of the DSL, another takes care of the concrete syntax and synthesizes the modelling environment, and the last one converts the provided sketches into models conformant to the induced meta-model. Next, we describe the main features of the *GraphicRepresentation* neutral meta-model and how it is used to produce a modelling environment for the DSL.

Fig. 8 shows the meta-model we have developed to represent graphical concrete syntaxes. It is an extended version of the one presented in [4], where we have added further features like layers, spatial relationships, reutilization through node inheritance, abstract nodes, and support for figures and edge styles.

Thus, we convert the concrete syntax information induced from sketches into this intermediate meta-model to be independent from the target technology, but

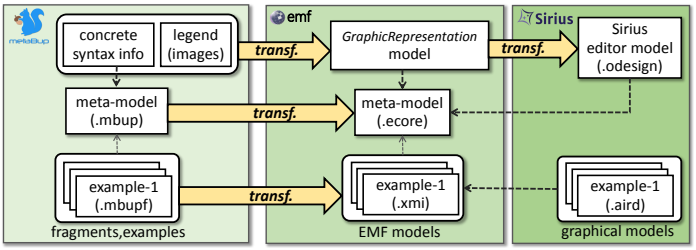


Fig. 7: Technical process: generating a (Sirius) graphical editor from examples.

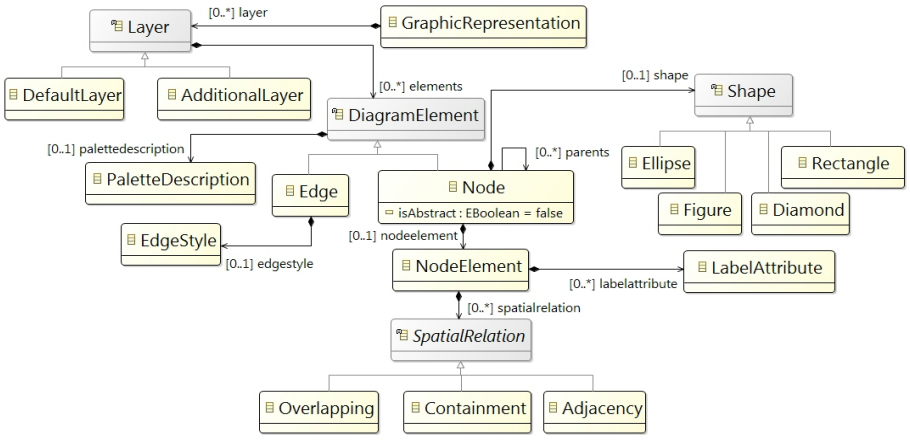


Fig. 8: Excerpt of the neutral *GraphicRepresentation* meta-model.

also, to be able to refine this information, e.g., by specifying palette information, organize elements in layers, or select labels for nodes. Graphical elements are organized into layers (abstract class Layer). A graphical representation has one DefaultLayer where all graphical elements belong by default, and zero or more AdditionalLayers. Layers contain graphical elements, which can be either Node-like or Edge-like. In both cases, they hold a PaletteDescription with information on how the element is to be shown in the palette. Nodes may be represented as geometrical shapes (Rectangle, Ellipse, etc.) or as image figures (class Figure). They can also display a label either inside or outside the node, being possible to configure its font style (class LabelAttribute). In addition, some nodes may need to be displayed in a relative position with respect to other nodes in the diagram, like being adjacent to (class Adjacency) or being contained in (class Containment) other nodes. Edges can specify a line style like solid, dash, dot or dash-dot (class EdgeStyle). Finally, we enable the reuse of graphical properties by means of relation parents and attribute isAbstract in class Node, so that graphical properties defined for a node are inherited by its children nodes.

The generation of the modelling environment requires establishing a correspondence between the abstract syntax meta-model of the DSL and the concrete

syntax meta-model in Fig. 8. Node-like elements have a direct correspondence (e.g., meta-classes are mapped to a Node and a Shape). References are mapped into Edges, while their concrete syntax annotations are mapped into an EdgeStyle. Both Nodes and Edges keep a cross-reference to the corresponding class or reference in the abstract syntax meta-model (omitted in the figure). In addition, if the references are annotated with @containment, @adjacency or @overlapping, they get assigned a Containment, Adjacency or Overlapping object respectively. All created elements are included in the default layer and receive a PaletteDescription.

To generate the modelling environment, we first synthesize an ecore meta-model with the definition of the DSL abstract syntax, and then, we transform the obtained *GraphicRepresentation* model into a Sirius model (*.odesign) describing the graphical syntax and its correspondence to the ecore meta-model. This latter transformation is implemented using ATL.

## 6 Tool support

The architecture of our solution encompasses the drawing tool yED, and two Eclipse plug-ins: *metaBup* [12] and *EMF Splitter* [4]. While *metaBup* supports the whole bottom-up abstract syntax construction process, we provide a specific *metaBup* exporter that wraps the resulting meta-model and passes it to *EMF Splitter*, which produces a fully operational graphical modelling environment from it. In the following, we explain how these two tools are integrated to support the presented approach, as well as the extensibility mechanisms of the tools.

### 6.1 Tool support for the generation process

Domain experts can create sketches with yED as shown in Fig. 9. Once an initial set of examples is ready, the modelling expert creates a new *metaBup* project. This will initially contain a blank meta-model file with *mbup* extension, and empty *fragments* and *legend* folders. The yED sketches are imported one by one, and converted into text fragment models in the shell console of *metaBup*. Once parsed, the modelling expert can modify the fragments if needed. The revised fragments are fed to the meta-model induction process, which may trigger refactorings on the meta-model. Fig. 10 shows the tool once the sketch of Fig. 9 has been parsed, and the current meta-model (accessible on the second tab of the editor). Technically, we need to copy the images used in the yED palette (right side of Fig. 9) into our *legend* folder.

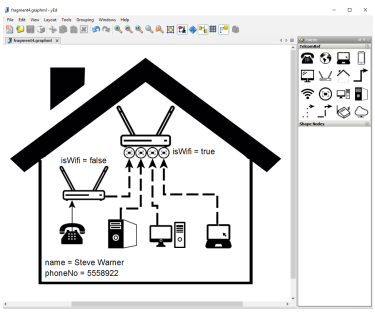


Fig. 9: Sketch drawn in yED.

After each iteration (i.e., addition of a fragment), a text version of the drawing is stored in the *fragments* folder of the project. These fragments are validated

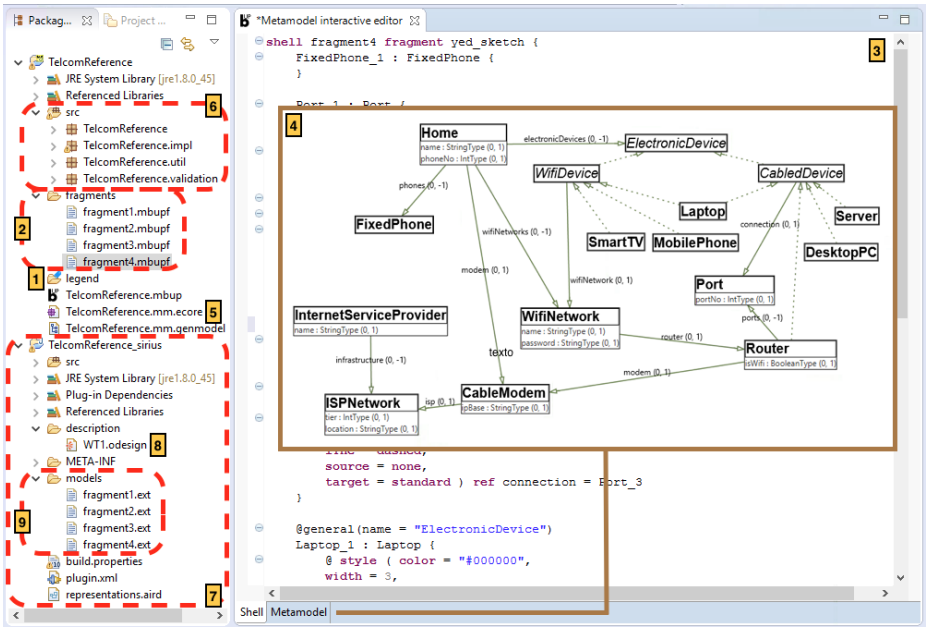


Fig. 10: *metaBup* tool: (1) Legend folder, (2) Fragments folder, (3) Parsed sketch in textual format, (4) Current version of meta-model, (5) Generated Ecore meta-model, (6) Java code generated from Ecore meta-model, (7) Generated Sirius project, (8) Sirius editor model, (9) Models transformed from the initial sketches.

upon each meta-model change, so that they will be error-flagged if they become inconsistent after a meta-model modification.

After processing all sketches, the modelling expert can produce the Sirius-based editor by just clicking on a button. In this way, first some necessary EMF artefacts are automatically generated, like the *ecore* and *genmodel* files (label 5 in Fig. 10), and the generated meta-model Java classes (label 6). These resources contain the equivalent representation to our working meta-model in EMF. The modelling expert is prompted to type a file extension for the models built with the new editor ("ext" in our example).

Then, a new Sirius *Viewpoint Specification* project is automatically created by internally using *EMF Splitter* (label 7 in Fig. 10). This created project includes two key elements: (i) an *odesign* file, the core resource of a Sirius editor, describing the DSL concrete syntax and its mapping to the DSL abstract syntax, and (ii) a folder named *models* containing models equivalent to those in the *fragments* folder, but now in *xmi* format. These files actually serve as validation units, since they are expected to be represented in the new editor similarly to the original sketches. The generated Sirius project can then be run, and Fig. 11 shows the resulting editor with one model coming from an initial sketch.

Altogether, for the running example, we synthesized a graphical DSL using 4 fragments, with 13 object types, 4 edge styles and using 3 spatial relationships (containment, overlapping and adjacency, but not alignment). The system au-

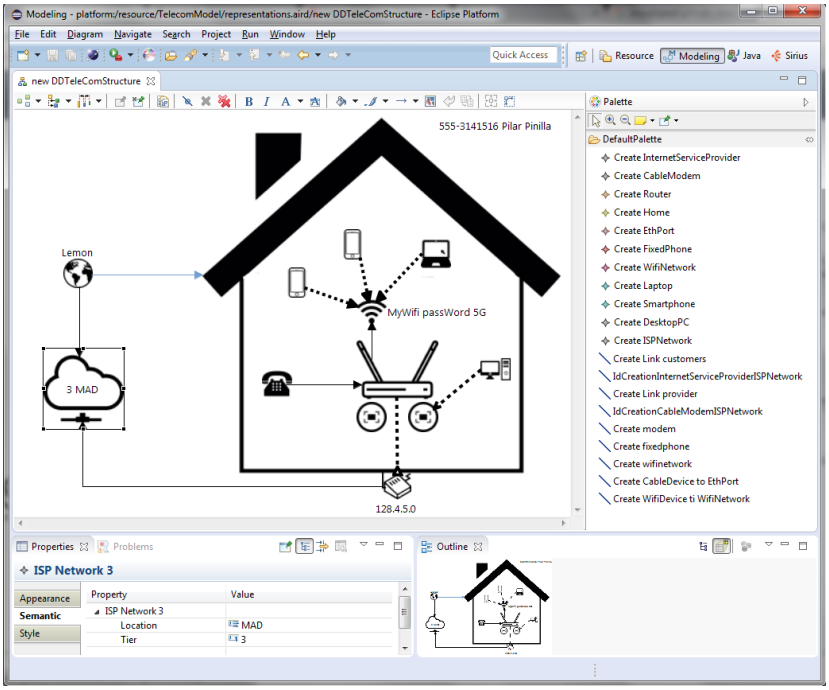


Fig. 11: Sirius graphical modelling environment for the running example.

tomatically induced a meta-model with 16 classes, 16 attributes, 13 references and 8 inheritance relationships. Finally, the generated Sirius *odesign* model contains 178 objects. The details of this case study, and some other examples, are available at `http://miso.es/tools/metaBUP.html`.

### 6.2 Extension mechanisms

Our tools can be extended (via Eclipse extension points) in different parts of the process, as shown in Fig. 12. First, there is the possibility to contribute new fragment importers (label *1*). For this purpose, we provide a platform-independent "pivot" meta-model to represent sketch information [12], from which we produce the internal textual representation shown in the paper. We currently have importers from Dia and yED, but other drawing tools could be supported as well. Additionally, we provide a meta-model for modelling the graphical properties explained in Section 4 (see Fig. 4). As spatial relationships between objects are automatically inferred from fragments, it is necessary to save object locations (attributes width, height, x and y in Fig. 4).

New meta-model refactorings can be added to *metaBup* (label *2* in Fig. 12). As the meta-model grows, the modelling expert is suggested suitable refactorings to be performed on the meta-model. We natively cover basic rules like pluralizing multi-target reference names or generalizing common features to abstract classes, but also give the chance to create custom meta-model modifications [12]. The

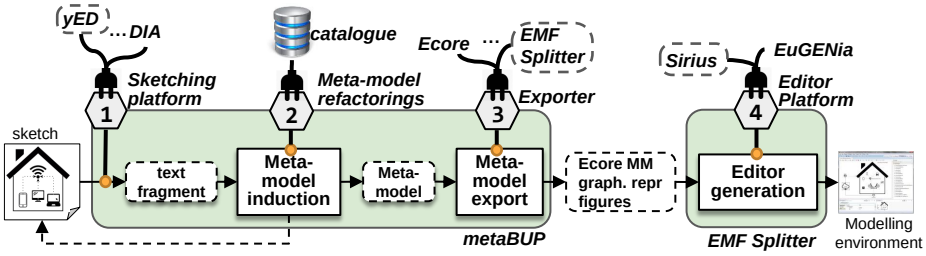


Fig. 12: Extension points: (1) Sketching platform, (2) Meta-model refactorings, (3) Exporter, (4) Editor platform.

tool can also be extended with meta-model exporters (label *3*), like the one we have presented for *EMF Splitter*. Finally, *EMF Splitter* currently targets the generation of Sirius-based editors, but other technologies like EuGENia could also be targeted (label 4 in the figure).

## 7 Related work

While MDE is founded on the ability to process models with a precisely defined syntax, some authors have recognised the need for more flexible and informal ways of modelling. This is useful in the early phases of system design [14, 16, 18], or as a means to promote an active role of domain experts in DSL development [2, 19], as we advocate in this paper. Next, we review works aiming at both goals.

There are two orthogonal design choices enabling flexible modelling in DSL development: (i) the use of examples to drive the construction process, and (ii) the explicit generation of a meta-model and a modelling tool different from the drawing tool used to build the initial examples.

Regarding the first design choice, "by-demonstration" techniques have been applied to several MDE artefacts, like model transformations [8], but their use is not so common to describe graphical modelling environments. The closest work to ours is [2], which describes a system atop Microsoft Visio to derive DSLs by demonstration. Given a single example, the system derives the concrete syntax from the icons in the palette, and some abstract syntax constraints, e.g., concerning the connectivity of elements. This information is recorded and used within Microsoft Visio. Instead, we derive an explicit meta-model, infer spatial relationships like containment and overlapping, and generate a modelling tool. Moreover, our induced meta-model supports modelling concepts like abstract classes, inheritance, compositions and attributes, which are not found in [2].

The approach in [19] uses yED to draw examples of the DSL. Types are assigned to elements on the basis of labels, and some predefined functions check for shape overlapping, colour or proximity. All modelling is performed within yED, and no meta-model or dedicated modelling environment are generated.

We believe that creating a meta-model and a modelling environment on top of a meta-modelling framework has some benefits. First, it guides the user in filling slot values, which otherwise should be done via tags in a diagramming tool

like Visio. Moreover, slots and links have a type, which enables type-checking. Second, the created models can be manipulated by standard model management languages for model transformation or code generation.

Some tools for DSL development are based on generating an external modelling tool. For instance, EuGENia live [15] is a tool for designing graphical DSLs that runs on the browser. The tool supports on-the-fly meta-model editing while the user is editing a sample model and its concrete syntax. The tool can export an Ecore meta-model enriched with concrete syntax annotations, which can be used to generate an Eclipse GMF-based environment.

Finally, some modelling tools promote flexibility in the early phases of system design by offering sketching capabilities similar to pen-and-paper drawing. For instance, SKETCH [16] provides an API to enable sketch-based editing on Eclipse. Calico [14] is a sketching tool designed for electronic whiteboards, where the sketched elements can be scrapped and reused in other parts of the diagrams. FlexiSketch [18] derives simple meta-models from sketches, but the extracted meta-model does not support conceptual modelling elements like class inheritance, abstract classes or different association types (e.g., compositions).

Altogether, our approach is novel as it enables the creation of graphical DSL editors based on drawings produced by domain experts, generating a meta-model and a dedicated modelling environment. This approach helps in transitioning from informal modelling in a diagrammatic tool, to formal modelling in a modelling tool, where models are amenable to automated manipulation.

## 8 Conclusions and future work

This paper has presented our approach to the example-based generation of graphical modelling environments. In our approach, domain experts contribute with sketches built with diagramming tools, and our system induces a meta-model and a graphical modelling environment, currently based on Sirius. The paper has shown the advantages of the approach, like: (i) there is no need to code or create editor specifications; (ii) it lowers the barrier to build graphical environments, which is a highly technical task requiring expert knowledge; (iii) it bridges the gap between drawing tools (likely used by domain experts in early phases of the development) and modelling tools (useful for automated model manipulation); and (iv) drawings can be transformed into models and be manipulated using MDE technology (transformations and code generators).

In the future, we plan to perform a user study to evaluate the construction process and the generated editor. To facilitate the validation of the final editor by the domain experts, we plan to integrate our mmXtens language [13], which is able to generate "interesting" example models using constraint solving. We also plan to improve our support for the editor evolution. For instance, a common scenario might be the manual modification of the Sirius editor model. To avoid overriding these manual changes, we may employ techniques similar to [11], where manual changes are described as a program that is reapplied when re-generation occurs.

**Acknowledgements.** Work supported by the Spanish Ministry of Economy and Competitivity (TIN2014-52129-R), the Madrid Region (S2013/ICE-3006), and the EU Commission (FP7-ICT-2013-10, #611125).

## References

1. 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.
2. H. Cho, J. G. Gray, and E. Syriani. Creating visual domain-specific modeling languages from end-user demonstration. In *MiSE@ICSE*, pages 22–28, 2012.
3. J. de Lara and H. Vangheluwe. AToM$^3$: A tool for multi-formalism and meta-modelling. In *FASE*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.
4. A. Garmendia, A. Pescador, E. Guerra, and J. de Lara. Towards the generation of graphical modelling environments aided by patterns. In *SLATE*, volume 563 of *CCIS*, pages 160–168. Springer, 2015.
5. Graphiti. `https://eclipse.org/graphiti/`.
6. R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.
7. J. Hutchinson, J. Whittle, and M. Rouncefield. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Sci. Comput. Program.*, 89:144–161, 2014.
8. G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer. Model transformation by-example: A survey of the first wave. In *Conceptual Modelling and Its Theo. Foundations*, volume 7260 of *LNCS*, pages 197–215. Springer, 2012.
9. S. Kelly and R. Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.
10. S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.
11. 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 Part I*, volume 6394 of *LNCS*, pages 211–225. Springer, 2010.
12. J. J. López-Fernández, J. S. Cuadrado, E. Guerra, and J. de Lara. Example-driven meta-model development. *Software and System Modeling*, 14(4):1323–1347, 2015.
13. J. J. López-Fernández, E. Guerra, and J. de Lara. Example-based validation of domain-specific visual languages. In *SLE*, pages 101–112. ACM, 2015.
14. N. Mangano, A. Baker, M. Dempsey, E. O. Navarro, and A. van der Hoek. Software design sketching with calico. In *ASE*, pages 23–32. ACM, 2010.
15. L. M. Rose, D. S. Kolovos, and R. F. Paige. Eugenia live: A flexible graphical modelling tool. In *XM*, pages 15–20. ACM, 2012.
16. U. B. Sangiorgi and S. D. Barbosa. SKETCH: Modeling using freehand drawing in eclipse graphical editors. In *FlexiTools @ ICSE*, 2010.
17. Sirius. `https://eclipse.org/sirius/`.
18. D. Wuest, N. Seyff, and M. Glinz. Flexisketch team: Collaborative sketching and notation creation on the fly. In *ICSE*, volume 2, pages 685–688, 2015.
19. A. Zolotas, D. S. Kolovos, N. D. Matragkas, and R. F. Paige. Assigning semantics to graphical concrete syntaxes. In *XM@MoDELS*, volume 1239 of *CEUR Workshop Proceedings*, pages 12–21. CEUR-WS.org, 2014.