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

An example is worth a thousand words: Creating graphical modelling environments by example

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

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

Received: date / Revised version: date

Abstract Domain-Specific Languages (DSLs) are heavily used in model-driven and end-user development approaches. Compared to general-purpose 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 a rather passive role in their development and hinders a wider adoption of graphical DSLs.

Our aim is achieving a simpler DSL construction process where domain experts can contribute actively. For this purpose, 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 synthesizes 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 and adjacency. Our system, called metaBUP, is implemented as an Eclipse plugin. In this paper, we demonstrate its usage on a running example in the home networking domain, and evaluate its suitability for the construction of graphical modelling environments by means of a user study.

Key words Domain-Specific Modelling Languages, Graphical Modelling Environments, Example-Based Meta-Modelling, Flexible Modelling.

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

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) that provide high-level primitives tailored to a particular field [19]. Hence, MDE projects frequently need to create DSLs and their associated modelling environments. DSLs are also heavily used in end-user development approaches [28] in order to allow users with no or little computer science background to perform small programming tasks in particular domains using DSLs.

The concrete syntax of a DSL may be graphical or textual, though in this paper we focus on graphical DSLs [26]. Many tools have emerged along the years to build environments for graphical DSLs [10,15–17,26,32, 40,48]. However, building such environments remains a technical, complex and time-consuming task. For example, developing a graphical editor with Graphiti [15] requires manual programming based on a large Java API. In the case of GMF [16] and Sirius [48], it is necessary to describe the different aspects of the editor by building one or more models, which may become very detailed, large and hard to build and maintain for non-experts, and which frequently 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 in a second stage. This approach hinders the active participation of domain experts in the DSL construction process, who might find it easier to work with examples rather than with meta-models [3,14,34] 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, as otherwise, they may reject the resulting DSL [14,25].

To avoid these obstacles, 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 graphical examples built by domain experts using drawing tools like PowerPoint, Dia or yED. Our framework processes the provided examples to derive a meta-model by using the bottom-up techniques presented in [34], and it also extracts a description of the graphical concrete syntax that includes graphical forms for classes (*svg* files), edge styles, and spatial relationships like containment or adjacency. This information is used to synthesize a graphical modelling environment that mimics the graphical syntax used in the examples, and 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., model transformations and code generators). As a result, a graphical DSL environment is generated with no need to code or create complex technical specifications, hence, giving rise to the motto of the paper title *"an example is worth a thousand words"*. Our proposal is backed by a working prototype, called metaBUP, available as an Eclipse plugin at `http://miso.es/tools/metaBUP.html`.

This is an extended version of our previous paper [35] with the following contributions. Prominently, we have performed a user study where eleven participants, playing the role of domain experts, have used our tool to develop a graphical modelling environment from examples. From the obtained results, we outline some lessons learnt, best practices and common pitfalls. In addition, we now provide a more detailed description of the approach to extract the concrete syntax from examples, and give heuristics to handle overlapping or conflicting spatial relationships coming from different examples. The comparison with related works has also been improved to make it more systematic, and grounded on the basis of the running example. Finally, we have added a section with a gallery of DSL examples and a discussion of capabilities and limitations of the approach.

The remainder of this paper is organised as follows. First, Section 2 presents an overview of our approach and a running example. Then, Section 3 introduces example-based meta-modelling, and 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 our tool support. Next, Section 7 discusses the results of our user study. Section 8 presents a gallery of DSL examples, and discusses the kinds of language our method is suitable for. Finally, Section 9 compares our approach with related research, and Section 10 draws some conclusions and lines of future work. The appendices contain the documents and questionnaires used in our user study, and detail the effort spent by the modelling expert.

2 Overview and running example

Our approach permits the automatic generation of graphical modelling environments from examples (i.e., from drawings made with informal diagramming tools). This includes the automatic derivation of the abstract and concrete syntax of the targeted DSL, as Figure 1 shows. While the process is fully automated, some intervention is allowed, e.g., to customise the names of the extracted relations, or to supervise meta-model refactorings. We term our approach *"what you draw is what you get"* (WYDIWYG), as the resulting environment mimics the provided examples. However, one obtains all the advantages of a *modelling* environment, like type checking, constraint evaluation, and the possibility to apply model transformations or code generators to the created models.

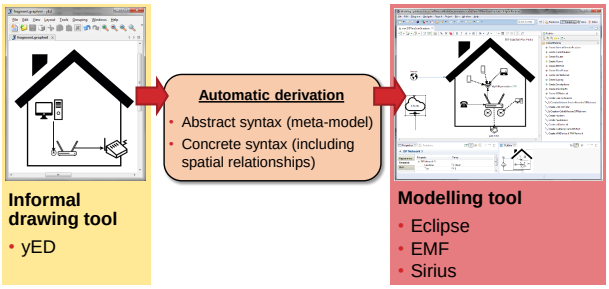


Fig. 1: From examples to modelling environments.

Figure 2 shows a more detailed view of the process for our example-based generation of graphical modelling environments. It involves two roles: the *Domain expert*, and the *Modelling expert*. The former has knowledge in the domain where the DSL is to be built, and is responsible for providing graphical examples and ultimately validating the generated environment. In this way, domain experts provide domain knowledge for the DSL by means of examples instead of working at the meta-model level, which could be challenging for them as they may lack the necessary computer science background. They can define the examples using publicly available drawing tools (e.g., yED), and may require discussing to identify good examples. We currently do not provide support for monitoring such discussions, helping with the selection of suitable domain type names and icons, or deciding the slots each object in the examples should have. On the other hand, the modelling expert has experience in modelling and meta-modelling, and on the specific MDE platform being used to create the DSL. Moreover, the task of this role is monitoring the meta-model derivation process from which the desired DSL environment is derived.

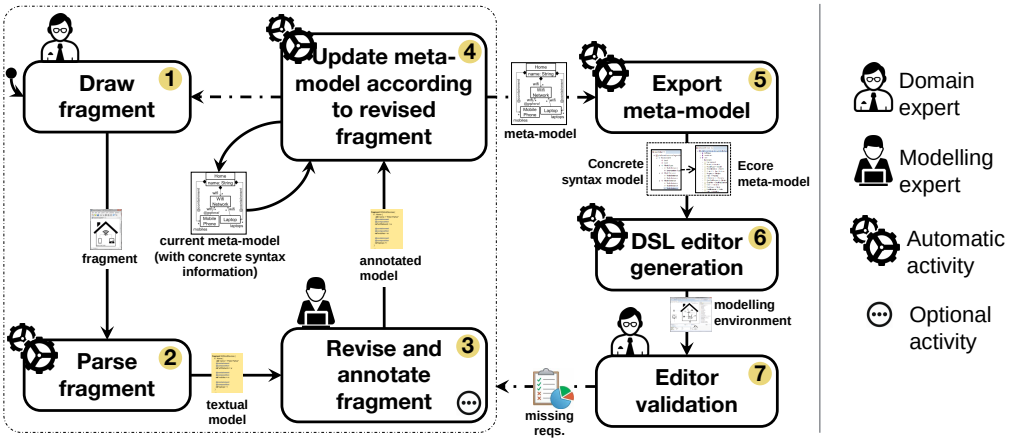


Fig. 2: Example-based process for developing graphical DSLs.

The core part of our process (steps 1-4 in Figure 2) is iterative. Here, the domain expert provides input examples made with tools like yED, portraying how models should look like graphically (label 1). These examples may represent complete models, or they may focus on an aspect of interest and therefore be partial, in which case we call them *fragments*. Then, the examples are automatically parsed into models conforming to our own internal meta-model, which are more amenable to manipulation (label 2). These models are represented textually and declare the existing objects, attributes and relations in the examples. Moreover, they are enriched with annotations that make explicit information regarding the graphical rendering of the elements in the examples (e.g., spatial relationships between objects or line styles). The modelling expert can optionally edit this textual representation (label 3) to set more appropriate names to the derived relations, or to trigger refactorings in the meta-model derivation process that takes place next (label 4). In this automatic derivation process, the meta-model is updated so that it accepts the provided example. Moreover, the meta-model elements become automatically annotated with the concrete syntax information (i.e., icons, edge styles, spatial relationships) extracted from the examples. This way, an iteration step finishes when the meta-model under construction evolves to accept the revised fragment. This process can be repeated with new fragments, which would update the previously derived meta-model accordingly.

After the system processes all provided examples, the modelling expert can export the derived meta-model to a suitable format. In our current implementation, we generate an Ecore [54] meta-model for the abstract syntax, and the concrete syntax information is exported in the form of a model that annotates the Ecore meta-model (label 5). Then, the editor generator can be invoked to obtain a fully operating editor that mimics 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), likely based on the converted examples, requirement documents, or using languages tailored to the validation and verification of meta-models and DSLs [36,37]. If necessary, the domain expert can refine the DSL by providing further examples and re-generating the editor.

2.1 Running example

As a running example to illustrate our approach, we will develop a DSL in the home networking domain. The DSL is inspired by one of the case studies in the Sirius gallery[1]. In this DSL, we would like to represent the customer data held by internet service providers (ISPs), 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 WiFi-enabled) router to which the landline phone is connected, and with a number of Ethernet cable ports. WiFi networks are password protected and work in a frequency range. Moreover, each home may have both cabled devices (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, Figure 3 shows one fragment built with yED[2], 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 ISP network, and the name of the ISP. The legend to the right assigns a name to every picture used in the drawing.

[1] https://eclipse.org/sirius/gallery.html
[2] https://www.yworks.com/products/yed

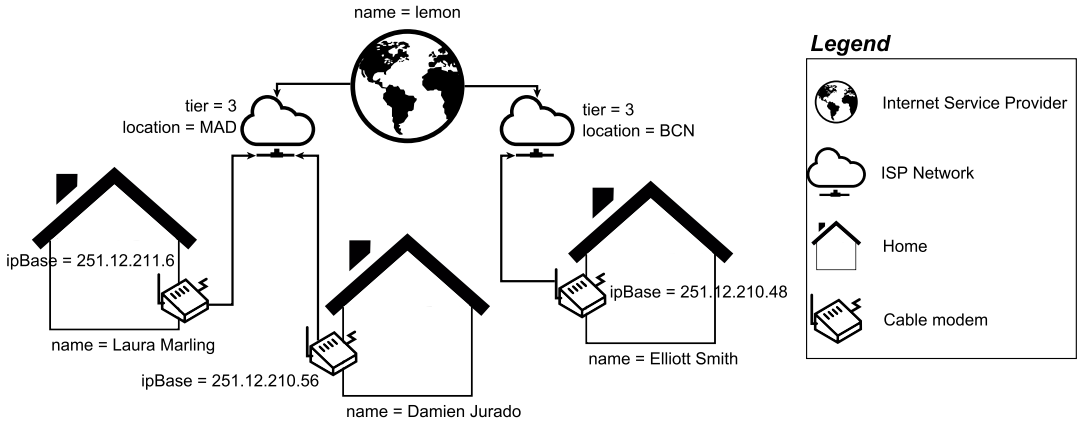


Fig. 3: Fragment showing a connection between customer homes and an ISP.

3 Example-based meta-modelling

In [34], we introduced a bottom-up meta-modelling technique that enables the automatic derivation of a meta-model starting from sketches[3] built using drawing tools. The main idea is to start at the object level (sketches) instead of at the class level (meta-models). The rationale is two-fold. On the one hand, domain experts with little or no background on computer science might find it more familiar working with examples than with generalizations. This is so as, in daily life, people are confronted with *exemplars* (of animals, houses...), while meta-models contain *universals*, i.e., abstract generalizations of concrete examples. On the other hand, modelling tools can be too rigid for domain experts, who might prefer the freedom of drawing tools and use the graphical notation most intuitive to them. Hence, domain experts keep working at the object level by sketching examples, and our tool generalizes these examples into a meta-model.

The meta-model derivation process starts by parsing the provided sketch or fragment into a textual internal representation that is easier to manipulate by the modelling expert. The type of the parsed objects is obtained from a legend that assigns a name to each symbol in the fragment, as shown in Figure 3. Fragments have an *open-world* semantics, in the sense that they only convey the relevant information for the given scenario, and thus, they may omit additional information that will be given in further fragments (e.g., they may miss attributes or relations). As explained in Section 2, examples are a special kind of fragments used to represent complete models, and they have a *closed-world* semantics as they need to be correct when evaluated as full-fledged models.

For instance, Listing 1 shows the textual representation model automatically obtained from parsing the fragment in Figure 3. 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. The fact that this is a fragment (in opposition to an example) is indicated with the keyword fragment in line 1.

```
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 Figure 3.

[3] 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.

The modelling expert can revise the textual fragment to annotate its objects, slots and links. The annotations can provide design or domain information accounting for well-formedness constraints of the DSL (see [34]), or they can convey concrete syntax details. In addition, the fragment importer automatically adds some concrete syntax annotations documenting the link styles and spatial relationships between the objects in the graphical fragment. Node icons or shapes do not need to be recorded in annotations as they are already provided by the legend.

As an example, the importer added the annotation @overlapping in lines 4, 14 and 20 of Listing 1. They annotate the modem references declared by three Home objects in lines 6, 16 and 22, respectively. In this way, these annotations convey the fact that Home and CableModem objects overlap each other. We will detail the use of this kind of annotations in Section 4. In [34], we reported on another use of annotations, as a means to encode meta-model integrity constraints. This is the case of the @composition annotation in lines 5, 15 and 21. As we will see in Section 4, these @composition annotations were heuristically added due to the existence of overlapping.

Textual fragments contain directed links, which are translated into directed references in the meta-model (in contrast to associations). The direction of the link is taken from the direction of the graphical edge in the fragment, or from the spatial relationships using some heuristics that are described in Section 4. In any case, a link can be set to result in a bidirectional association (i.e., two opposite references) in the meta-model, by annotating it with @bidirectional [34].

Once the modelling expert has revised the textual fragment, this is automatically processed to derive an appropriate meta-model that "accepts" the fragment, or to evolve a previous version of the meta-model if it already exists. For example, if a fragment contains objects of an unknown type, this type is incorporated into the meta-model. Similarly, if an object assigns a value to features that are not present in its type, then its meta-class is extended with these new features.

Figure 4 shows the meta-model derived from the fragment in Listing 1. 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 assign a type to 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. We natively support numeric (int and float), String and boolean data types. Regarding the cardinality of references, the modelling expert can configure their default lower bound (0 or the minimum in the fragment) and upper bound (unbounded or the maximum in the fragment). In this example, the default value for the lower and upper bounds is 0 and unbounded, respectively. Hence, references will be assigned an upper bound * as soon as an object points to two or more objects using edges with the same style (e.g., infrastructure), or 1 if it points to at most 1 object (e.g., modem). Moreover, if two references have the same name, stem from objects with the same or a compatible type, and point to 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. Finally, domain and concrete syntax annotations are transferred from the fragment to the appropriate meta-model element (e.g., @overlapping). For clarity, in this and the following figures containing meta-models, we represent the @composition annotation using the standard black-diamond notation (see, e.g., reference modem).

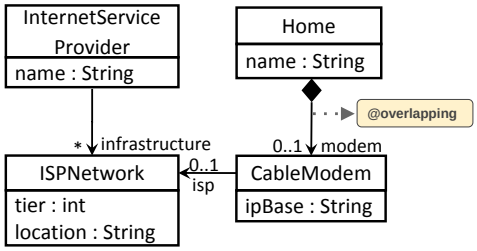


Fig. 4: Meta-model derived from the fragment in Listing 1.

The system is equipped with an assistant that may recommend refactorings to improve the quality of the meta-model that results from processing each fragment. The catalogue of available refactorings is extensible. As an 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 [34].

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 sketched fragments which are no longer passive documentation, but they are used to derive a meta-model. Up to now, our technique had been only able to derive the abstract syntax of the DSL [34]. 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). We also provide an evaluation of this contribution in Section 7.

4 Example-based concrete syntax inference

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

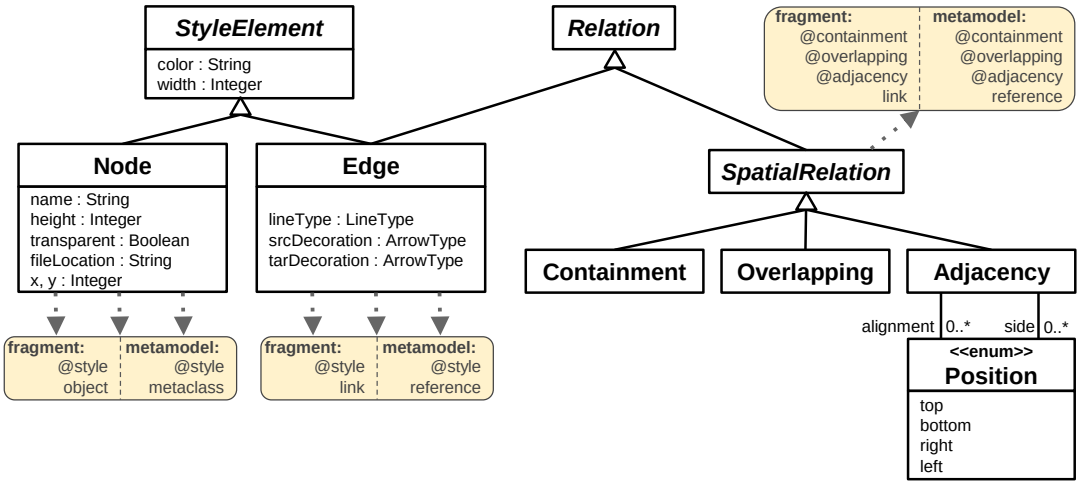


Fig. 5: Graphical properties inferable from fragments, and corresponding annotations.

Figure 5 shows a conceptual model with the graphical properties that we automatically extract from fragments and use to derive the concrete syntax of the DSL. Some are explicit features from the elements in the drawing, like their colour or size. Other properties are implicit relationships concerning the relative position of icons, like overlapping or adjacency. Both kinds of graphical properties are encoded as annotations of the corresponding objects and links in the textual representation of the fragment. Then, these annotations are transferred to the appropriate domain meta-model classes and references when the fragment is processed. Figure 5 also shows the correspondence between the graphical properties and the elements they can annotate.

In the remainder of this section, 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 fragments, 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 in 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, Figure 6 shows to the right the *legend* folder that contains the *svg* files used to represent each domain object in the fragment to its left. The file names (Home, CableModem, etc.) will be used as type names for the objects represented using the icons in the fragment.

Regarding edges, we detect and classify their style, recording their colour, line width, style (e.g., dotted, dashed) and source and target decorations. Other types of decorations, like labels or decorators in the middle of the edge, are not supported.

As an example, Figure 6 contains an edge linking a Router and a Cable modem. When the fragment is imported, the link is annotated with the identified style (lines 27–29 in Listing 2), and its name is made of the concatenation of the graphical features of the style. For instance, the name inferred for the link is not modem, but the one struck through (see lines 30–31 in Listing 2). Because the text fragments can be edited, the modelling expert 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 reference 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 existing reference modem.

By taking the edge style as a source of information, two links between the same two objects will have the same type if their style coincides, and a different type if their style is different. This avoids *symbol overload* for link types [41]. Nonetheless, the modelling expert can deactivate this functionality if the edge style is irrelevant for the domain. In such a case, any link between the same two objects will be assigned the same type, and it will be named using the type of the link's target object in lowercase (cablemodem in case of the link in line 31).

Any graphical information annotation on a link will be transferred to the corresponding meta-model reference, and eventually, to the concrete syntax generator. Meta-classes, on the other hand, never carry graphical information with them, since we store their exact representation (their icon) in the *legend* folder.

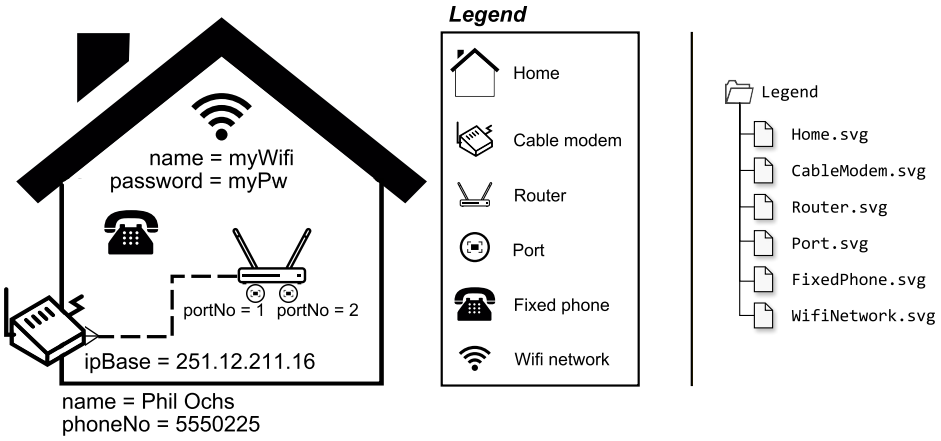


Fig. 6: Fragment with spatial features (left). Content of the *legend* folder (right).

```
1  fragment fragment2 {
2    home1 : Home {
3      attr phoneNo = 5550225
4      attr name = "Phil Ochs"
5
6      @overlapping
7      @composition
8      ref modem = cableModem1
9
10     @containment
11     @composition
12     ref electronicDevices = router1
13
14     @containment
15     @composition
16     ref phones = fixedPhone1
17
18     @containment
19     @composition
20     ref wifiNetworks = wifiNetwork1
21   }
22   router1 : Router {
23     @composition
24     @adjacency(side = bottom)
25     ref ports = port1, port2
26
27     @style ( color = "#000000", width = 3,
28         line = dashed, source = none,
29         target = crows-foot-many )
30     ref '00000_3_dashed_none_crows-foot-many'
31         modem = cableModem1
32   }
33   fixedPhone1 : FixedPhone { }
34   wifiNetwork1 : WifiNetwork {
35     attr name = "myWifi"
36     attr password = "myPw"
37   }
38   port1 : Port { attr portNo = 2 }
39   port2 : Port { attr portNo = 1 }
40   cableModem1 : CableModem {
41     attr ipBase = "251.12.211.16"
42   }
43 }
```

Listing 2: Textual representation of the fragment in Figure 6.

4.2 Detection of spatial relationships

Sometimes, spatial relationships between graphical objects have a meaning in the domain [52], which needs to be reflected in the meta-model. However, it is likely that the domain expert is unaware of whether a certain layout implies some requirement for the domain. For this reason, we automatically detect spatial relationships in fragments, and let the modelling expert discard them by editing the textual fragments. By keeping them, they will be reified in the meta-model as references. We currently identify and give support to three kinds of spatial relationships:

- *Containment*: a graphical object is within the bounds of another object.
- *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 sides from which objects are attached to each other (e.g., two objects adjacent left-to-right), and if in addition they are aligned and how (e.g., at the bottom).
- *Overlapping*: two graphical objects are superimposed but not contained.

When we detect one of these spatial relationships, we represent it explicitly as a reference in the meta-model. In the case of containment, the reference is added to the container and points to the containee. For adjacency and overlapping, we use the following 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, and if all objects have the same size, there is the possibility to make the reference bidirectional. 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 Figure 6 illustrates the three kinds of spatial relationships, which are automatically detected when the fragment is imported (see Listing 2). In the fragment, the Home contains a Router, a Fixed phone and a Wifi network; 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 fragment; hence, a new link annotated as @overlapping in the textual representation (line 8) is created from the node with the bigger icon (the Home), to

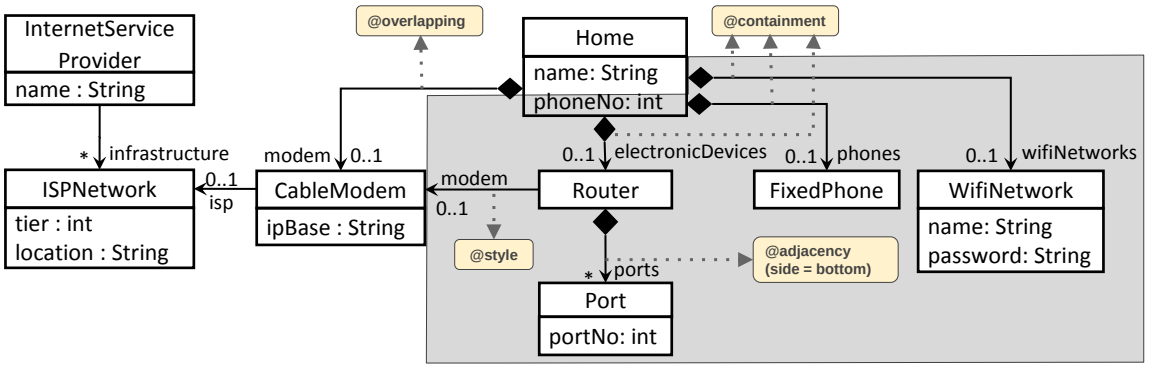


Fig. 7: Updated meta-model after processing the fragments of Listings 1 and 2.

the smaller one (the Cable modem). Finally, the Router has two adjacent Ports to the bottom side in the fragment; since there are multiple ports, the Router is added a link annotated as @adjacency in the textual representation (line 25). The side parameter of this annotation indicates the side where the adjacency occurs (at the bottom of the router), but it can be removed if this 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 23 in Listing 2). This helps in organizing and realising only a minimal but *sufficient* set of meta-model references, in the sense that it suffices to capture all inferred spatial relationships. For example, both Ports in the fragment are contained in the Home, but this relation is not made explicit in the textual representation because they are already adjacent to the Router, which is inside the Home. Hence, the importer adds the @composition annotation, which allows inferring that Ports are indirectly contained in Home objects via Router objects (line 11). Section 4.3 will analyse this and other heuristics that we have devised to handle redundant relationships and conflicts.

Figure 7 shows the resulting meta-model after automatically processing this second fragment, including the annotations for style properties and spatial relationships. The grey-shaded elements are new with respect to Figure 4. In particular, the meta-model has been extended with four new classes: Router, FixedPhone, WifiNetwork and Port. Each class has been added attributes accounting for the slots appearing in the fragment objects. As an example, the Home class now includes the new attribute phoneNo that appears in the second fragment but not in the first one. Regarding the new references added, all compositions correspond to detected spatial relationships, while the non-composition Router.modem represents the edge from the router to the cable modem in the provided fragment. The lower bound of the created references is 0 because this is the default value set by the modelling expert, who can change it if appropriate. Moreover, if the modelling expert modifies the derived reference cardinalities, all previously imported fragments would be checked in real-time to notify potential inconsistencies with the current meta-model version.

4.3 Resolution of conflicts in spatial relationships

In this section, we introduce several heuristics to avoid representing redundant spatial information, to deal with multiple spatial relationships converging on the same objects, and to handle optionality of spatial relationships appearing in some fragments but not in others. We will illustrate these heuristics showing small excerpts of fragments and the meta-models inferred from them. In general, our heuristics represent spatial relationships as compositions in order to identify objects that are part of another ones, and to facilitate exporting the generated meta-model to frameworks like EMF, where each object should be contained directly or indirectly in a root object. While the default behaviour of some heuristics can be configured, in all cases, the modelling expert can modify the obtained result if it does not fit the domain at hand.

4.3.1 Avoiding redundancy As previously mentioned, our system tries to create the minimum set of references needed to represent all spatial relationships discovered in an imported fragment. For this purpose, it takes advantage of the transitivity of composition relations to decide which ones encode redundant information and can be removed.

Figure 8 illustrates the situations where redundant relations can be safely removed. We consider the cases where a containment relation can be derived out of other overlapping, adjacent or containment relations.

In case a), A contains objects of type B and C, which in their turn overlap. Moreover, B is bigger than C, and therefore, our heuristic identifies C objects as parts of B objects. In the generated meta-model to the right, we create a composition reference from A to B due to the containment, and another from B to C due to the overlapping and the fact that the B object is bigger. However, although the C object is also contained in A, we

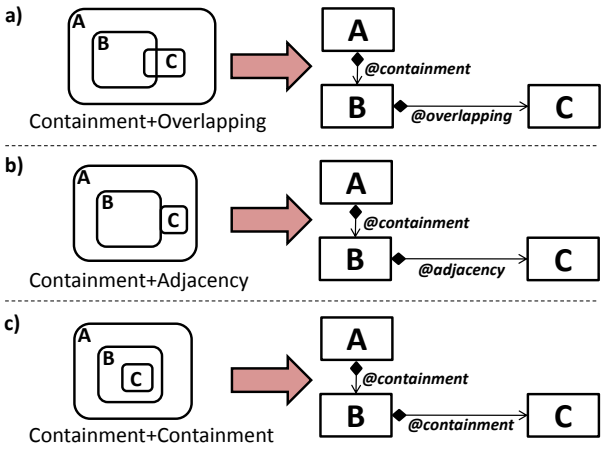


Fig. 8: Avoiding the creation of redundant references to express containment.

do not add a composition reference between classes A and C because the other two composition references already imply that C objects are indirectly contained in A objects.

The situation in case b) is similar, where the adjacent object C is identified as a part of B. Case b) occurs in the running example, as the Home object in Figure 6 contains a Router with adjacent Ports. In this case, a composition reference is created between Home and Router, and between Router and Port, but not between Home and Port. This situation is frequent in languages where nodes have "ports" or "attachment points" to connect to other elements (called plex languages [11]) and where nodes can be nested. Component diagrams are an example of this kind of languages, which we will illustrate in Section 8.3.

Finally, in case c), we use the transitivity property of spatial containment, so that if C is inside B, and B inside A, then C is inside A; hence, we do not reify the latter spatial relationship with a composition because it is implied by the former relations.

4.3.2 Multiple spatial relationships Fragments where a set of objects participate in more than one spatial relationship between each other, may lead to alternative ways to arrange the composition relations in the derived meta-model to avoid conflicts between them. Figure 9 illustrates some representative scenarios.

In scenario a), containment and overlapping relationships arise for an object of type B. In particular, the B object is contained in an A object and overlaps with a C object, but in contrast to case a) in Figure 8, the A object does not contain the C object but both overlap. As a result, the inferred meta-model has two references to represent the different overlappings of C, and the question is which one of them should be a composition, and which one should not. Declaring that a reference is a composition implies a stronger relation between the related classes. Due to the semantics of composition, both references cannot be compositions because that would not allow an object of type C to be contained in both references at the same type, which is the case shown in the figure (the C object overlaps with objects of type A and B simultaneously). Removing one of the references would not capture this case either. Hence, both references are needed, and the modelling expert should decide whether the composition corresponds to the reference defined either by class A (option 1) or class B (option 2). The possibility to always use the container (class A) or the containee (class B) as holder of the composition can be configured by default.

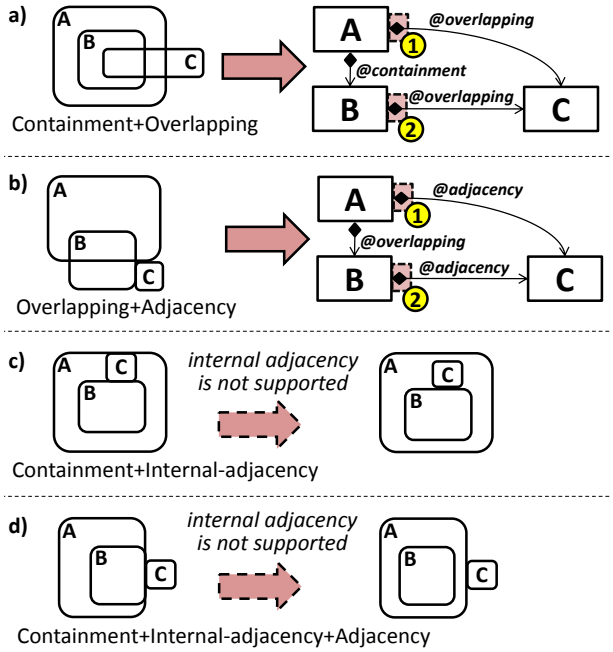


Fig. 9: Handling multiple spatial relationships converging on the same objects.

Scenario b) is similar to the previous one, where objects of types A and B overlap and are adjacent to a C object. Assuming that a composition reference is created from class A to B because the A object is bigger than the B object, the question is again which one of the two references inferred from the adjacency relationships should be a composition. As in scenario a), the default behaviour can be customised.

The last two scenarios show unsupported combinations of spatial relationships. In scenario c), objects of types B and C are contained in an object of type A, and moreover, the C object is adjacent to both A and B objects. However, the adjacency of objects A and C is internal[4]. Our system does not currently support this kind of adjacency, which is just interpreted as containment, as shown in the fragment to its right. Therefore, this scenario is treated like case b) in Figure 8. Similarly, in scenario d) of Figure 9, the internal adjacency between objects of types A and B is interpreted as containment,

[4] By internal adjacency we mean an object that is simultaneously contained by and adjacent to one of the inner borders of another object.

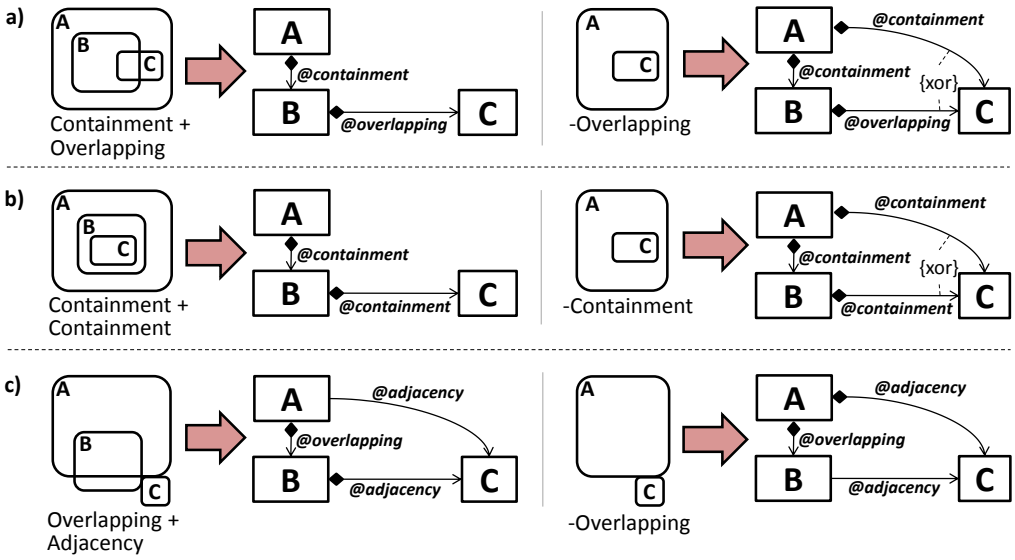


Fig. 10: Handling optionality of spatial relationships.

and hence, the fragments shown to the left and to the right are equivalent.

4.3.3 Optional spatial relationships Special cases also arise when some spatial relationship is present in some *examples* but not in others, meaning that such relationship is optional. This may imply reifying previously derived spatial relationships as references, or reorganizing the composition references in the derived meta-model, as Figure 10 shows.

In case a) of Figure 10, there is a first fragment where the B and C objects overlap and are contained in an A object. Then, in a second fragment, a C object is inside an A object without overlapping with any B object. The meta-model obtained from the first fragment does not suffice to represent the containment relationship in the second fragment, since class A does not define a container reference for C objects. Hence, the meta-model is extended with a new composition from A to C, together with a xor constraint to indicate that C objects can be contained either in A objects (due to the containment relationship in the second fragment), or in B objects (due to the overlapping relationship in the first fragment)[5].

Similarly, in case b), C objects can be placed either inside of B or A objects. This is represented by two compositions from classes A and B to C, and a xor constraint.

In these two previous cases, the second fragment does not contain B objects, which makes the relation between B and C become optional because the fragment exemplifies that A objects can contain C objects directly (in addition to indirectly via B objects). Hence, the composition relation from A to C needs to be made explicit.

Finally, in case c), the first fragment includes overlapping objects of types A and B, which in addition are adjacent to a C object. In this case, the modelling expert has selected the second option in case b) of Figure 9 to infer the meta-model, and hence, the reference from B to C has been marked as a composition. Then, in the second fragment, there is a C object that is not adjacent to any B object. Therefore, we move the composition from the reference between B and C, to the reference between A and C. This change would not be necessary if the modelling expert had selected this option when processing the first fragment.

5 Generation of graphical modelling environments

Our approach to synthesize the graphical editor proceeds in two steps, both automatically performed using model transformations. First, we convert the information gathered from the fragments into a technology-neutral representation, and then, this representation is translated into a technology-specific editor specification. We currently target Sirius [48], but other technologies like EuGENia [32] could be easily targeted as well.

Figure 11 outlines this process, where three transformations take place: one generates the meta-model with the abstract syntax of the DSL (label 1); another takes care of the concrete syntax by constructing a technology-neutral model with the graphical information (label 2) from which a modelling environment for a specific technology is synthesized (label 3); the last transformation converts the provided fragments into models conformant to the derived meta-model (label 4). Note that, as explained in the previous section, our approach represents the concrete syntax information extracted from fragments, including spatial relationships, as meta-model annotations (boxes *concrete syntax info* and *legend (images)* in Figure 11). Then, the transformation with label

[5] Although this xor constraint would be implicit in UML, we show it explicitly for clarity.

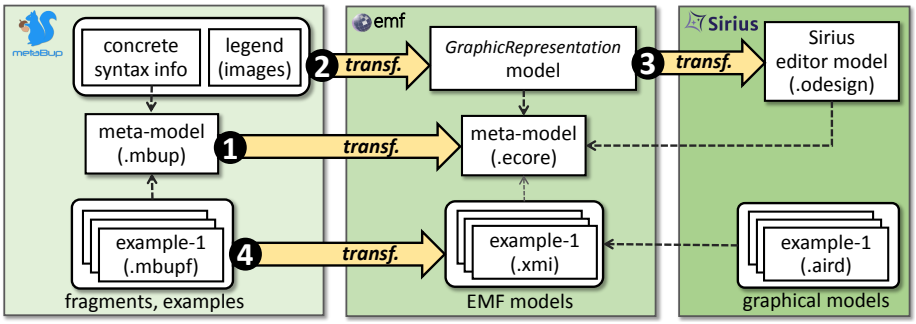


Fig. 11: Technical process: generating a (Sirius) graphical editor from examples.

2 collects this information and creates a graphical representation model where, e.g., spatial relationships are reified as objects. Next, we describe in more detail this transformation dealing with the concrete syntax, since it is the most challenging.

Figure 12 shows an excerpt of the neutral meta-model we have developed to represent graphical concrete syntaxes, which is called *GraphicRepresentation*. It is an extended version of the meta-model presented in [12], 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 derived from fragments into this intermediate meta-model to be independent from the target technology, but also, to be able to refine this information, e.g., by specifying palette information, organize elements in layers, or select labels for nodes.

A graphical representation in our *GraphicRepresentation* meta-model is organized into one or more layers (class Layer). One of them is the defaultLayer where all diagram elements belong by default.

DiagramElementTypes define the graphical representation types of the objects of a certain meta-model class, and can be visualized either as nodes (class NodeType) or edges (class EdgeClassType). In both cases, they may hold a PaletteDescription with information on how the element is to be shown in the editor palette. NodeTypes may be represented as geometrical shapes (Ellipse, Rectangle, etc.) or as image figures (class Figure). They can display a label either inside or outside the node, being possible to configure its font style (class LabelAttribute). Moreover, 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. The container, overlapping or adjacent node types are indicated through reference SpatialRelation.with. On the other hand, as abovementioned, classes can also be visualized as edges using class EdgeClassType. In such a case, it is possible to configure their style (class EdgeStyle). Regarding the representation of references, they can be visualized as links by means of the class EdgeReference, and can define a style and decorators (omitted in the figure).

The *GraphicRepresentation* meta-model also enables the reuse of graphical property definitions by means of relation inheritsFrom in class NodeType, so that graphical properties defined for a node are inherited by its children nodes. If a node is only being used as a placeholder for reusable properties but is not intended for being drawn on its own, then it should be marked as abstract.

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 Figure 12. We consider abstract syntax meta-models defined with Ecore, for which mappings can be established according to Figure 13. In particular, classes in the domain meta-model (class EClass) can be represented either as nodes (class NodeType) or as edges (class EdgeClassType), and are referred to through the reference DiagramElementType.eClass. In case the class is represented as an edge, it is possible to configure the references of the class acting as source and target of the edge. Attributes in the domain meta-model (class EAttribute) can be mapped into a LabelAttribute. References in the domain meta-model (class EReference) can be mapped into EdgeReferences, and their concrete syntax annotations are mapped into an EdgeStyle. In addition, if a reference is annotated with @containment, @adjacency or @overlapping, then it gets assigned a Containment, Adjacency or Overlapping object, respectively (not shown in Figure 13). All created graphical elements are included in the default layer and receive a palette description.

Altogether, 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 the Atlas Transformation Language (ATL) [21].

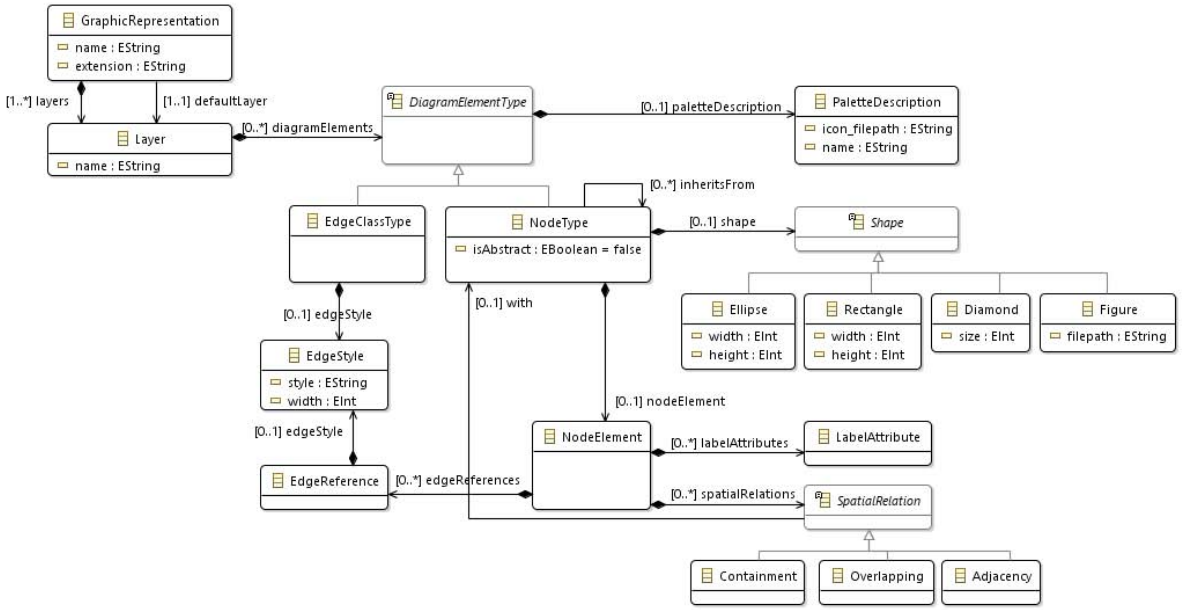


Fig. 12: Excerpt of the *GraphicRepresentation* meta-model.

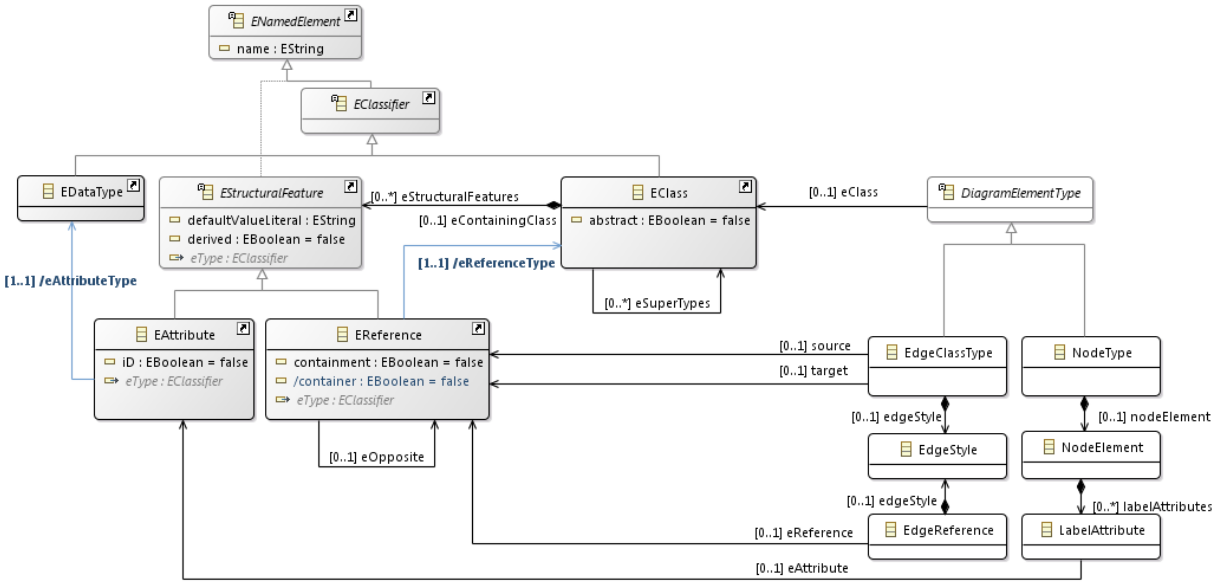


Fig. 13: Mapping between *GraphicRepresentation* meta-model and Ecore meta-model (classes of Ecore are shaded).

6 Tool support

The architecture of our solution encompasses drawing tool like yED or Dia to draw the graphical fragments, and two Eclipse plug-ins: metaBup [34] and *EMF Splitter* [12,20]. 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 (Section 6.1), as well as the extensibility mechanisms of the tools (Section 6.2).

6.1 Tool support for the generation process

Domain experts can draw fragments with yED as shown in Figure 14. 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 drawings are imported one by one, and automatically converted into text fragment models, which then are shown in the shell console of metaBup. Once parsed,

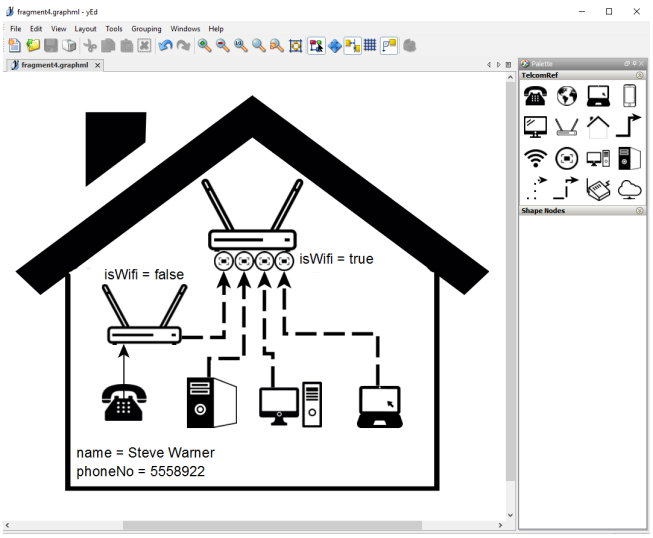


Fig. 14: Fragment drawn in yED.

the modelling expert can modify the textual fragments if needed. The revised fragments are fed to the meta-model derivation process, which may trigger refactorings on the meta-model. Figure 15 shows the tool once the fragment of Figure 14 has been parsed, including the current version of the derived 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 Figure 14) into our *legend* folder.

Each time the tool processes a fragment, it stores a text version of the drawing in the *fragments* folder of the project. These fragments are validated upon each meta-model change, so that they will be error-flagged if they become inconsistent after a meta-model modification.

After all fragments have been processed, the modelling expert can produce the Sirius-based modelling 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 Figure 15), 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 that will be built with the new modelling editor ("ext" in our example).

Then, a new Sirius *Viewpoint Specification* project is automatically created by internally using *EMF Splitter* (label 7 in Figure 15). 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 serve as validation units, since they are expected to be represented in the new editor similarly to the original fragments. The generated Sirius project can then be run, and Figure 16 shows the resulting editor with one model coming from an initial yED fragment.

Compared to the yED editor, the synthesized environment relies on an underlying meta-model, and can perform validation of integrity constraints (e.g., cardinalities, OCL) and check whether the values of object slots are conformant to their data type. Moreover, since the generated environment is an Eclipse plugin, it can work with the rich Eclipse ecosystem of MDE tools, e.g., for model transformation or code generation.

The environment can be evolved if so desired by providing new fragments, and regenerating it again. This means that, if the generated editor definition is modified by hand, those changes will be overridden. In future work, we plan to add better evolution support to avoid this overriding.

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 automatically produced a meta-model with 16 classes, 16 attributes, 13 references and 8 inheritance relationships. 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

Both metaBup and *EMF Splitter* can be extended via Eclipse extension points in four different parts of the process, as shown in Figure 17. 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 the objects and relations in fragments [34], from which we produce the internal textual representation shown in the paper. We currently have importers from yED and Dia, but other drawing tools could be supported as well. Additionally, we provide a meta-model for modelling the graphical properties explained in Section 4. Since spatial relationships are automatically inferred from fragments, it is necessary to save the location of objects in fragments (attributes width, height, x and y in Figure 5).

The catalogue of meta-model refactorings supported by metaBup is also extensible (label 2 in Figure 17). 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. These rules can be extended to create custom meta-model modifications [34].

The tool can also be extended with new meta-model exporters (label 3 in Figure 17), like the one we have presented for *EMF Splitter*. Similarly, *EMF Splitter* currently targets the generation of Sirius-based editors, but other technologies to build graphical editors like EuGENia could be targeted as well (label 4).

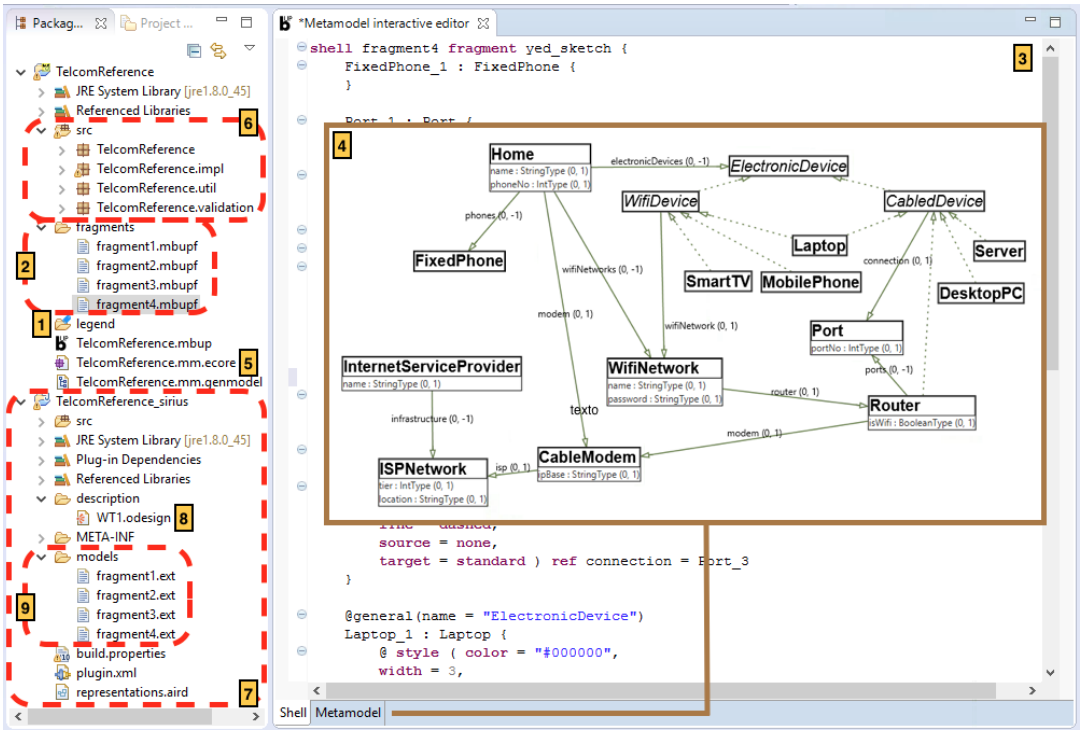


Fig. 15: metaBup tool: (1) Legend folder, (2) Fragments folder, (3) Parsed fragment in textual format, (4) Current version of meta-model, (5) Generated Ecore meta-model, (6) Java code generated from Ecore meta-model, (7) Generated Sirius project, (8) Sirius editor model, (9) Models transformed from the initial yED fragments.

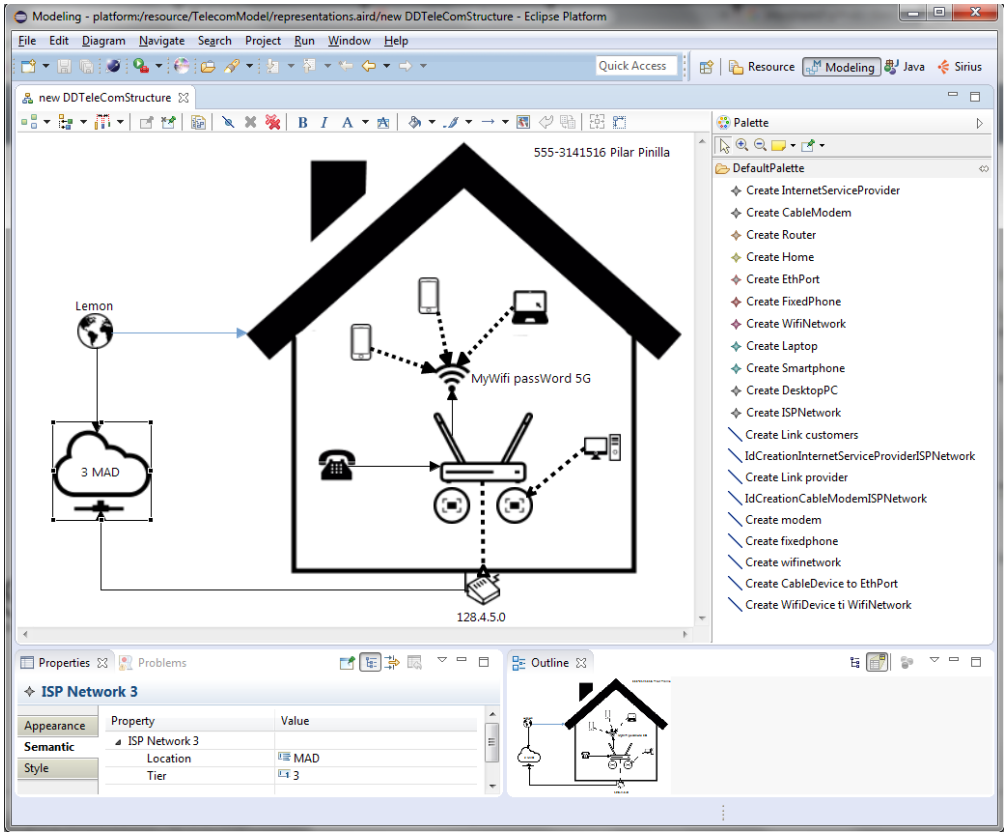


Fig. 16: Sirius graphical modelling environment for the running example.

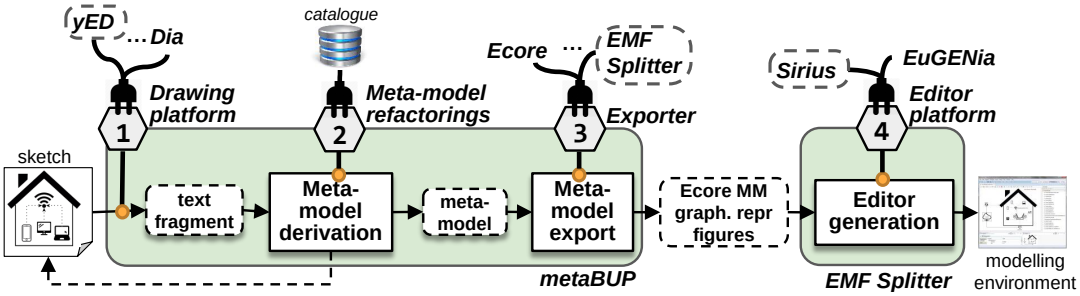


Fig. 17: Extension points: (1) Drawing platform, (2) Meta-model refactorings, (3) Exporter, (4) Editor platform.

On the other hand, the generated graphical editors are also extensible via the standard Eclipse extension points, e.g., to plug some MDE tools of the Eclipse ecosystem. However, we currently do not provide specific extension points for that.

7 Evaluation

We have conducted a user study to evaluate our example-based approach to generate graphical modelling environments. Since one of the goals of our proposal is enabling the active involvement of domain experts in the DSL environment construction process, the study was performed from the point of view of the domain expert. Hence, the participants in our study played the role of domain experts, whereas we (the authors) played the role of modelling experts. As domain experts, the participants were asked to provide fragments, as well as to evaluate the environments generated from them. In this way, the goal of our evaluation is two-fold: first, to assess whether our example-based approach is perceived as useful to generate graphical environments, and second, to explore to what extent the generated environments fulfil the domain experts' expectations regarding their devised DSL. Hence, our study explores the following two research questions:

- **RQ1**: How useful is our approach to create graphical environments?
- **RQ2**: How well do the generated environments reflect the devised DSLs?

From a technical perspective, we are also interested in the quality of the artefacts produced by our approach, which leads to a third research question:

- **RQ3**: How is the quality of the derived domain meta-models perceived?

7.1 Evaluation setup

To evaluate these research questions, we designed a user study that emulates a typical example-based workflow, with remote participants playing the role of domain experts (DEs), and the authors playing the role of modelling experts (MEs). Figure 18 summarizes this workflow. It includes the following six steps:

Step 1 (DE): provision of fragments. First, the participants were given an online textual description of the requirements of a DSL, and a yED installation which contained a palette with admissible icons for the DSL. We decided to use the DSL presented as a running example in this paper (i.e., home networks), as this would allow analysing whether the same requirements might lead to different graphical representations. Appendix 1 contains the provided description of DSL requirements, as well as the instructions to complete the experiment.

In this step, the participants used yED to draw as many examples as necessary to represent all desired aspects of the expected DSL, and uploaded these examples via a web application together with the time employed to complete them.

Step 2 (ME): generation of modelling environment. Starting from the fragments, we used our tooling to generate a graphical modelling environment for each participant. Then, we sent to each participant the environment generated out of his/her fragments. At this stage, we occasionally had to perform little formal corrections over the fragments, always ensuring a minimal intervention (see Section 7.4 for more details).

Step 3 (DE): evaluation of modelling environment. The participants were allowed to use the generated environment freely with no time restrictions. Then, they replied an online questionnaire rating different aspects, like resemblance of the generated DSL to their expectations, and remarkable or missed features in the modelling environment. We used both Likert scales with scores from 1 to 5, and free-answer questions. In case the participants had developed meta-models or modelling environments in the past, they were asked additional technical questions related to the quality of the generated meta-model, and could comment on their preferences on using an example-based meta-model or graphical editor construction process instead of using the typical top-down approach. The complete questionnaire is available in Appendix 1 (questionnaire 1).

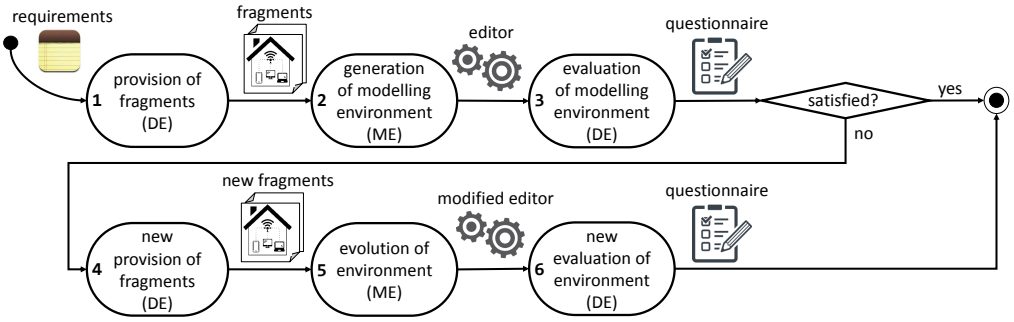


Fig. 18: Evaluation process involving domain experts (DE) and modelling experts (ME).

Step 4 (DE): new provision of fragments. Participants were given the opportunity to provide new examples complementing those in the first iteration. This was optional, only if they wanted to refine the generated environment, e.g., because they had spotted some defect on the environment, or because they had failed to represent some DSL requirement in the first iteration.

Step 5 (ME): evolution of environment. We used the new examples to evolve the initial version of the modelling environments.

Step 6 (DE): new evaluation of environment. The participants in this second iteration evaluated the new version of their editors, answering whether their quality had improved and which DSL aspects still remained uncovered. The complete questionnaire is available in Appendix 1 (questionnaire 2).

We invited 30 people with different backgrounds and ages to participate in the study. In total, 11 replied to our petition, 3 female and 8 male, with ages ranging from 24 to 46 years old. Amongst the different respondents, 8 were university employees (either in an academic or a technical position), 2 worked in the private sector (one in the IT field and the other in a different sector), and 1 was unemployed. Regarding their technical background, 6 of them had developed meta-models and graphical DSLs in the past, 2 had built meta-models but not graphical DSLs, 1 had used modelling languages like UML but had no experience on meta-modelling, and 2 had no experience on modelling or meta-modelling.

7.2 Evaluation results

This section shows the results of our evaluation. First, we analyse some features of the fragments provided by the participants. Then, we use this information as well as the questionnaire replies to give an answer to our three initial research questions.

7.2.1 Diversity of fragments Each participant could provide as many fragments as desired. Eventually, the number of provided fragments per participant ranges from 1 to 6, with a median of 2. Figure 19 shows how many participants (y axis) provided each number of fragments (x axis).

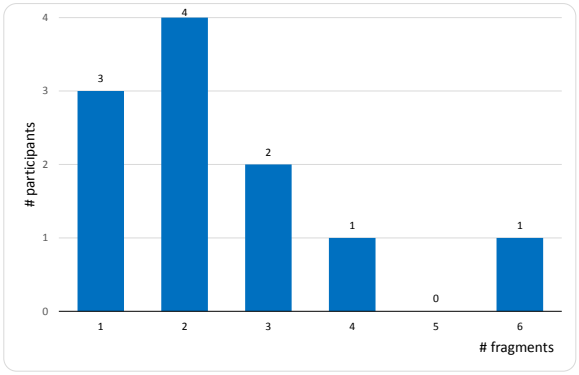


Fig. 19: Number of fragments per participant.

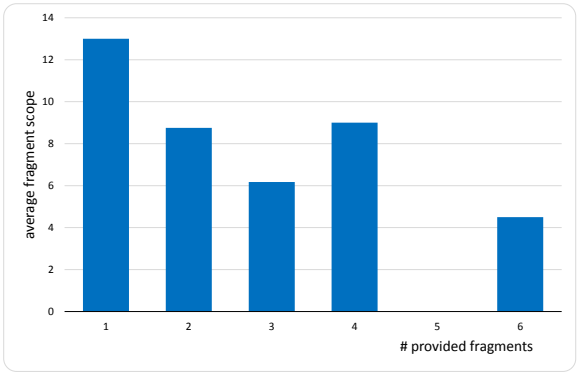


Fig. 20: Average fragment scope (i.e., number of element types) w.r.t. number of provided fragments.

We examine the structure of the provided fragments to assess the extent of use of the capabilities of our framework. First, we study the scope of each fragment. Similar to unit tests in test-driven development [6], in our methodology, each fragment is meant to identify a situation of interest (ideally one DSL requirement) using the minimal number of elements to convey the given meaning. The DSL palette for our experiment had 13 element types, and the average number of element types per fragment was 9 (see Figure 20). The three participants that provided a single fragment used all 13 element types in the fragment, which is understandable as, otherwise, their editors would have resulted incomplete.

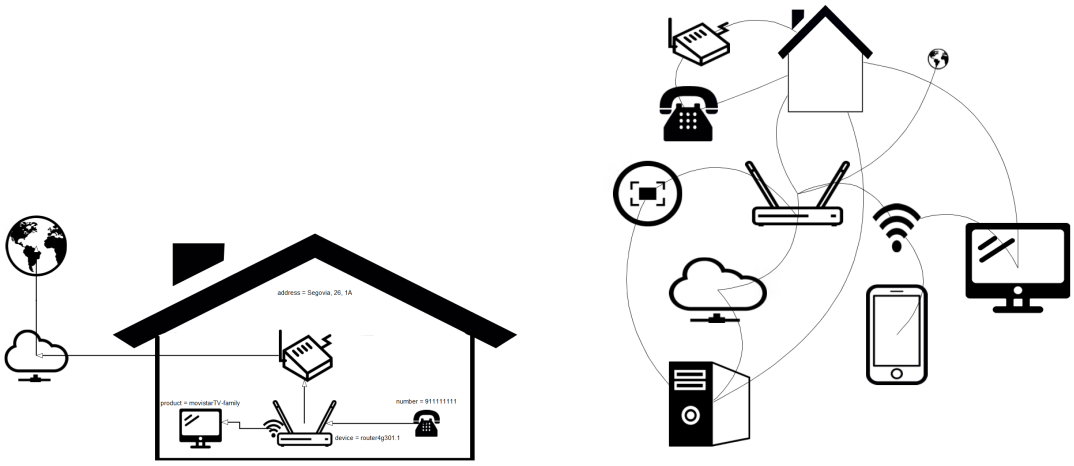


Fig. 21: User fragments with heavy (left) and meager (right) use of the supported graphical features.

Concerning size, fragments had an average of 12 objects and 9 edges, though their size strongly differ from 2 to 30 objects and from 0 to 29 edges. Considering that the average number of object types per fragment is 9, we observe low redundancy (i.e., few repeated objects of the same type).

If we compare the number of spatial relationships and edge-based relationships used in fragments, we find that objects are connected through edges 2,3 times more frequently than they are using spatial relationships (overlapping, adjacency or containment). While every participant used at least 1 edge per fragment, 4 participants did not employ any of the detectable spatial relationships. However, if we do not consider these 4 participants, the ratio spatial relationship/edge decreases to 1,3, which puts the average use of both kinds of relationships at a closer level. Still, the general shape of the DSLs was very much graph-like. Anyhow, it is remarkable that all participants with no modelling background made use of spatial relationships in their fragments.

Similarly, although the documentation that accompanied the DSL requirements detailed the possibility of using different edge styles, edge styling was seldom used. Only 2 out of the 11 participants exploited this option to discriminate different ways to connect pairs of the same object types. Just as illustration, Figure 21 shows to the left the fragment of a user who made heavy use of most of the graphical features supported by our framework, namely spatial relationships, edge styling and attribute labelling for nodes. The fragment to the right belongs to another user who merely connected the objects with non-styled edges and made no use of text labels.

Altogether, we can summarize the use of graphical features by the participants as follows:

- 100% used edges for connecting objects.
- 63% used spatial relationships.
- 27% used object or edge labelling.
- 18% gave style to edges for distinguishing different types of connections between pairs of objects.

Despite the DSL requirements document encouraged the use of edge styling and layout in fragments, and although the proposed problem really fostered their usage, the results evidence that, in the future, the possibilities of the environment should be further emphasized to potential users. On the other hand, we did not find a correlation between the technical background of the participants, and the number, size or structure of the fragments they provided.

Once we have analysed the features generally present in fragments, we answer the three research questions.

7.2.2 RQ1: How useful is our approach to create graphical environments? We consider that our approach is useful if it speeds up the construction of graphical environments and it promotes the active involvement of domain experts. To evaluate this, we first analyse the creation time of the environments in our study, and then, we assess their usability.

Using our approach, the time to create a graphical environment is the sum of the time employed to draw the DSL examples, plus the time required by the modelling expert to supervise the parsed fragments, plus the time to generate the environment from them. The supervision activities, which are detailed in Section 7.4 and Appendix 2, took in the order of a couple of minutes at most for every fragment. Generating the environment from the revised fragments is automatic, and its time negligible. Hence, the time to create a graphical environment is clearly dominated by the time to draw the examples. Figure 22 shows the time employed in this task by our 11 participants. The times range from 15 to 120 minutes, with a median of 37 minutes and an average of 43,8 minutes, while the time per fragment is between 3 and 60 minutes. Hence, the average time to create an environment for the DSL in our study was 43,8 minutes, with 72% of the participants employing even less. This time can be considered short, as developing a similar environment by hand would require implementing the

following artefacts (average numbers over all generated editors): an Ecore meta-model with 14 classes, 14 attributes, 22 references and 1,5 inheritance relationships; and a Sirius *odesign* model with 232 objects. Moreover, it would require having deep technical knowledge on these technologies, which probably domain experts would lack.

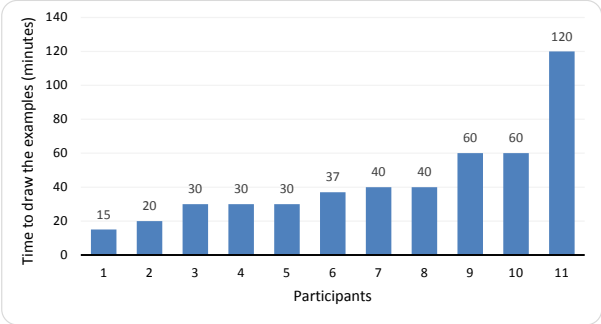


Fig. 22: Time employed to draw the fragments.

In the study, 6 participants had experience on developing meta-models and modelling environments. Surprisingly, three of them were the slowest (60, 60 and 120 minutes), while the other three were the fastest to build the examples (15, 20 and 30 minutes); hence, there is no correlation between time and MDE experience. The only 2 participants that had no experience on modelling or meta-modelling dedicated 37 and 40 minutes on drawing 3 and 1 fragments, respectively. This demonstrates that non-modelling experts can actively contribute to developing graphical editors by providing DSL examples, as our tool synthesizes working editors out of them.

Regarding the usability of the generated editors, Figure 23(b) shows how easy to use they are according to the participants. The answers range from average (3) to very easy (5), with a median of 4 (easy), and average of 4,1. These numbers suggest a good usability of the final environments, although the participants also mentioned some aspects to improve which are summarized in Table 1. In particular, one participant suggested reducing the high number of edge types that appear in the palette, e.g., by having one button for all of them, and deducing the type of any created edge from the types of the objects it connects. While this is a good strategy for large DSLs, the default drawing mode of Sirius is using a palette button per edge type, and hence, we plan to study the feasibility of this proposal in the future. The rest of suggestions are limitations of Sirius which we cannot overcome. As an example, two participants stated that containment seemed "odd" in the generated editor, and that objects placed in a container could not be moved to any other container. We should not ignore that the handling of containment is not currently one of the best fine-tuned features in Sirius. Regarding the editor aspects the participants liked the most (question Q9 in the questionnaire, see Appendix 1), they mentioned flexibility, simplicity, being easy to use, and the support of many edge styles. Table 2 details the answers to this question.

Table 1: Answers to question Q8: *Which aspects of the (generated) environment would you improve?*

There are too many edge types in the palette.
Creating new models is not intuitive.
Objects are hard to resize.
Handling of containment is intricate.
Moving an object between containers is not possible.
Difficulty to draw edges and layout.

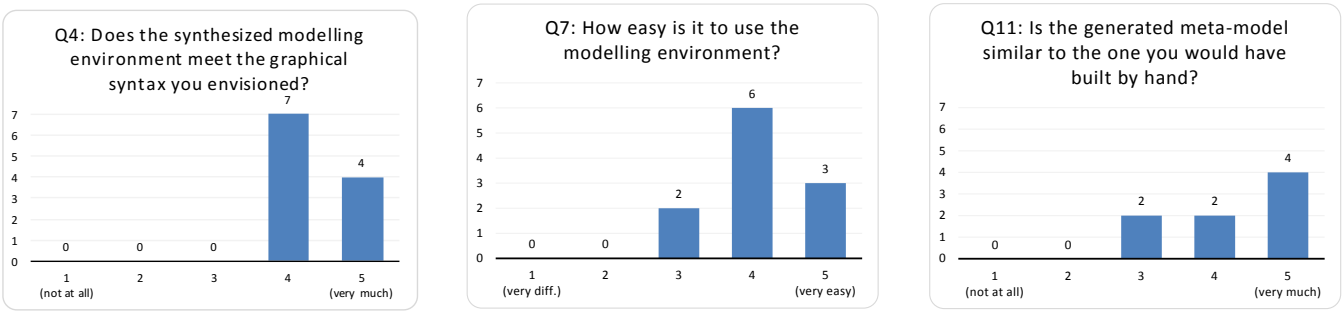
Table 2: Answers to question Q9: *Which aspects of the (generated) environment do you like the most?*

The conversion performed by the tool is spectacular.
Very intuitive and flexible.
The placement of the objects in the editor.
Easy to use.
Intuitive tool, which captures well the DSL semantics.
I liked the integration of a drawing tool within Eclipse.
Generating an editor out of 3 fragments is quite useful.

In summary, **the participants found our approach to create graphical modelling environments useful**. More importantly, participants with no modelling background were able to design a graphical DSL and, by providing examples, creating an editor for the DSL. Nonetheless, the participants have also proposed some improvements to the usability of the generated environments, which we plan to incorporate whenever possible in future versions of metaBup.

7.2.3 RQ2: How well do the generated environments reflect the devised DSLs? To answer this research question, we examine the responses to questions Q4, Q5 and Q6 in questionnaire 1. Question Q4 requests a score for grading how precisely metaBup was capable of producing a graphical syntax that resembles the original drawings. Figure 23(a) summarizes the given scores, which went from much (4) to very much (5), with a median of 4 and average of 4,36.

Questions Q5 and Q6 in the survey (both free-answer questions) provide more elaborate answers concerning the accuracy of the generated graphical DSLs. First, one participant complained that fixed and mobile phones could be placed both inside and outside homes in the generated environment. It is significant that this participant provided fragments in which phones were inside homes, and fragments in which they were not. Because we incorrectly interpreted these fragments as examples (i.e., as complete versions of models), our tool generated an environment where phones could be placed in



(a) (b) (c)

Fig. 23: Scores to different aspects of the generated environments and their underlying domain meta-models.

two different kinds of containers: homes and "the canvas". Interpreting the drawings as fragments (i.e., as possibly incomplete models) solves the problem. Alternatively, the environment could have been refined in a second iteration, though the participant deemed it was not necessary. Another participant stated that objects originally painted superimposed in the fragments had been substituted by containment relationships in the final editor. This is a limitation of Sirius, which does not support overlapping relationships between objects. To handle this, our Sirius exporter offers the possibility to choose which of the two remaining spatial relationships (adjacency or containment) should substitute overlapping in the editor. During the experiment, this preference was set to containment for all examples. Finally, two participants reported differences on the size and position of the model elements with respect to the original fragments, but they did not report mismatches regarding the expected DSL itself.

Among the aspects best captured by the generated DSLs, the users mentioned the edge styles and the containment and adjacency relationships. Anecdotally, one participant liked that the position of elements had been preserved in the migrated models, which surprisingly, was mentioned as an aspect to improve by another participant.

Notably, no participant requested a second iteration to refine the generated environments. This fact, and the average score 4,36 out of 5 (i.e., 93%) given by the participants when asked if the DSL met their expectations, indicate that **the environments reflect very well the devised DSLs**.

7.2.4 RQ3: How is the quality of the derived domain meta-models perceived? All solutions led to similar meta-models, with differences lying in how participants chose to graphically represent certain aspects of the domain.

Next, to answer RQ3, we analyse the replies to questions Q11, Q12 and Q13 in questionnaire 1, which were only available to participants with meta-modelling experience (8 out of 11 participants). Question Q11 provides a measure of the degree in which the derived meta-model matches the user's expectations. As Figure 23(c) shows, the similarity between the expected meta-model and the generated meta-model ranges from average to very much, with a median of 4,5 and average of 4,25. These numbers indicate that the derived meta-model was found similar to what a modelling expert would build by hand.

Question Q12 identifies aspects incorrectly captured by the derived meta-model, hence giving an indication of the perceived meta-model quality. This is of special interest in the case of participants that assigned lower scores to Q11. Table 3 summarizes the identified issues. Two participants commented that they would have created one abstract class holding common references to other classes. Interestingly, our tool supports this refactoring and recommended its application in these cases, though we did not apply it because we did not want to pervert the evaluation with manual modifications to the derived meta-model. Another participant complained that the name of some inferred references was strange, e.g., `containment`; as before, these names could have been modified by the modelling expert in the fragment revision phase. This same participant also missed some meta-model attributes to represent the object locations; however, this is not necessary in our approach as this information is directly managed by the concrete syntax layer, and anyhow, it would be easy to add it as a configuration option in the future. The remaining 5 participants (including one that ranked 3 to Q11) did not find errors in the derived meta-model. Overall, these results show that the participants perceived the quality of the derived meta-models as high.

Table 3: Answers to question Q12: *Which aspects does the (generated) meta-model not capture correctly?*

Common features are not generalised.
Missing attributes to represent object locations.
Some features have strange names, e.g., containment.

Regarding Q13, only 2 participants stated that they would prefer building the meta-model by hand, even though they had rated the derived and expected meta-models as very similar (maximum score in Q11). The remaining 6 participants would prefer using an example-based approach, 4 of them requiring the ability to modify the resulting meta-model by hand, and the other 2 considering this unnecessary.

Altogether, the participants ranked the quality of the generated meta-models as 4,25 out of 5 in average, hence, **the perceived quality of the generated meta-models is high**. Regarding the few detected issues (like the generalization of common features), our tool assists the modelling expert in their correction by recommending suitable refactorings.

7.3 Threats to validity

Next, we analyse the threats to the validity of our study.

We tried to minimise the selection bias by promoting the participation of people with different background, ranging from computer science students with a shallow knowledge of modelling techniques, professors both with and without expertise on DSLs, workers on technology companies without a specific training on modelling, and people working on non-technological companies. This is representative of very different domain expert profiles. However, the participants were not real experts on the selected home network domain, and therefore, they might have been less demanding when evaluating the expressiveness of the generated editors. We tried to minimize this effect by asking the participants whether the final DSL was the one they had in mind, for which the domain expertise is not relevant. Similarly, we did not offer any incentive (monetary or of any other kind) to the participants, which may have hindered their engagement leading to less accurate scores or answers, and preventing their participation on a second iteration of the editor [29].

Another threat to the internal validity of the results is the 6-7 days elapse since the participants provided the examples until they evaluated the generated environment, as in the meantime, some of their initial expectations regarding the DSL may have been distorted. This elapse was due to the variety of participant profiles, and in order to promote the participation, we granted 7 days to draw and submit the examples off-line, and another 7 to evaluate the generated editor and fill in the questionnaire. Moreover, 3 participants delayed their evaluation 7 extra days due to professional commitments.

On the other hand, having performed an off-line double-blind user study has eliminated any experimenter bias that we could have inadvertently introduced.

Regarding the generalizability of our results across people, as we mentioned before, we selected participants with different backgrounds on modelling and DSLs in order to make our results as general as possible. However, considering the nature of the proposed problem (in the field of computer network configuration), all of them had some training or working experience on computer science and programming. Hence, it remains as a threat to the external validity of our study considering other kinds of domain experts with a low technological profile.

Another threat to the generalizability is that the study is on the construction of one DSL, and hence, the results might be biased to the features of this DSL. To minimize this risk, we selected a DSL that allowed using a rich set of spatial relationships and connections between nodes (so-called hybrid visual languages [8]). To have an intuition of the generality of our approach, Section 8 explores the use of our framework to build a gallery of DSLs of different types.

Moreover, our study emulated a workflow where only one domain expert contributed DSL examples. Hence, our findings cannot be generalised to situations where contributions come from several experts who might even provide different representations for the expected DSL. In particular, in a project setting, there would probably be teams of domain experts providing fragments.

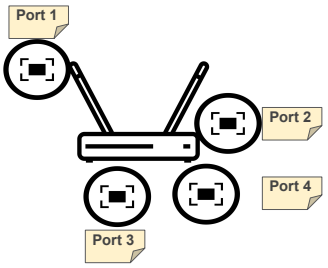
Finally, regardless the number of people invited to participate in our study, only 11 participants completed the evaluation. As stated in [29], recruiting participants in tool studies is difficult, but we aim at performing further studies with more participants, working in teams.

7.4 Discussion

Next, we discuss some interesting details of the experiment, lessons learnt, and open challenges for future work.

In our roles of modelling experts, we tried to interfere as little as possible in the editor generation process. However, in some cases, we had to perform little adjustments to the imported fragments to correct evident mistakes made by accident when drawing the examples, or to perform fixes that did not affect the semantics of the domain. We show an example in Figure 24. In yED, each element in the palette is contained in an invisible bounding box, and we calculate spatial relationships in accordance to this box. Thus, in Figure 24, although the intention of the domain expert was to draw all ports adjacent to the router, the two ports that are superimposed to the bounding box of the router (i.e., Port1 and Port2) get classified as overlapping references by our tool. Hence, in this case, we had to manually modify the imported fragment to delete the overlapping_ports reference and add Port1 and Port2 to the adjacency_ports reference.

Next, we list the adjustments we had to perform, indicating in parenthesis the frequency of changes with respect to the total number of participants (a detailed list of individual changes is given in Appendix 2):




```
1  fragment importedFragment {
2    router : Router {
3      @overlapping
4      ref overlapping_ports =
5        port1, port2
6
7      @adjacency
8      ref adjacent_ports =
9        port3, port4
10   }
11
12   port1 : Port {}
13   port2 : Port {}
14   port3 : Port {}
15   port4 : Port {}
16 }
```

Fig. 24: Common faux pas in the drawing of fragments (left) and its automatic parsing into text fragment (right).

- **Ignore invisible bounding box (3/11).** The abovementioned case, in which a participant is ignorant of the transparent bounding box of objects.
- **Rename reference (11/11).** Because fragments do not include reference names, we manually set reference names that were easily identifiable in the generated editor, in the format <*source*>2<*target*>.
- **Rename auto-generated superclass (1/11).** Our meta-model derivation algorithm is able to infer abstract superclasses for common features, assigning as class name a common substring of the children class names (e.g., Phone if the children classes are FixedPhone and MobilePhone). If the subclasses lack a common morpheme, the modelling expert is prompted to set a domain-significant name for the new class.

We found positive that, although the process entitles the modelling expert to perform deep changes over the fragments and the meta-model, little manual editing was needed to obtain well-valued editors.

Currently, our approach supports fragments from yED and Dia, and we based our evaluation on yED. To this respect, roughly half the participants expressed some discomfort with the use of yED. Most criticisms indicated the complexity to draw edges in yED as the main reason to delay the completion of fragments. However, when analysing the results of the survey, the time to draw each fragment does not seem significant in the assessment of the generated editors. Looking at Q4, which asked whether the synthesized environment met the envisioned graphical syntax, the average score given by the participants who took 24 minutes (the average time per fragment) or more to complete each fragment is 4,25 out of 5, while the score given by the participants who took less than 24 minutes to draw each fragment is 4,43, which is not a significant difference. Probably, a more usable or popular drawing tool, like PowerPoint, would have led to shorter drawing times. Anyhow, our framework can be extended with importers for other drawing tools, not being the particular selection of drawing tool a limitation of our approach.

Similarly, most identified deficiencies regarding the usability of the generated editors are due to limitations of some Sirius features. Although our framework is likewise extensible in the export stage as it is in the import, Sirius is one of the most powerful tools nowadays for developing graphical modelling editors.

Regarding the fragment provision process, we have detected that there is a need to better instruct domain experts in some features of our framework, in particular concerning the usage of spatial relationships and edge styling. Moreover, some participants did not understand the difference between fragment and example. A better understanding would have helped in clarifying certain ambiguities and misinterpretations, contributing to the utter completion of more qualified editors. In our framework, examples are built in the same way as fragments, but only the former represent complete models. Hence, the meta-model derivation algorithm does not have to apply heuristics or prompt disambiguation tasks to the modelling expert when processing examples, as it may happen for fragments. Ideally, fragments should contain minimal sets of objects representing portions of domain information, whereas examples are more widespread. In both cases, they can be used as test cases [37] to identify conflicts that should be resolved before altering the domain meta-model.

Finally, the experiment has purposely omitted some advanced features of our system for simplicity. For instance, yED drawings can be annotated to introduce domain restrictions which get compiled into OCL meta-model constraints (see [34] for more details). Actually, one participant stated in the questionnaire that the Sirius editor should have incorporated an OCL constraint.

8 Gallery of DSLs

This section presents additional DSL examples built with our tool, and identifies some of the strengths and limitations of our approach. The examples are also available at `http://jesusjlopezf.github.io/metaBup/gallery.html`.

We base our presentation on the classification of visual languages proposed in [8], which distinguishes two main types of visual languages: *connection-based* and *spatially defined*. The former type includes plex-like languages (where nodes are connected via ports) and graph-like languages, further divided into multipartite graphs (with different types of nodes) and hypergraphs (where edges can connect any number of nodes). Spatially defined languages are classified into containment-based and adjacency-based. Grid-based languages (e.g., a chess board) belong to both categories as board cells contain pieces and are adjacent to each other. Finally, *hybrid* languages, like statecharts, have features of both connection-based and spatially defined languages.

As it can be observed, our running example is hybrid as it makes use of graph-like connections (e.g., routers

connected to different devices), containment (devices inside the home), adjacency (ports adjacent to the router) and overlapping (modems overlapping the home).

Next, we present examples of DSLs in every category. The research question we aim to answer is whether our framework can handle different types of DSLs – according to a well-established classification of visual features – in order to identify strengths and limitations. Hence, we did not rely on domain experts to build these examples, but we built them ourselves.

8.1 Connection-based languages

Figure 25 shows an editor built with our tooling for a graph-like language to define fault-tree diagrams. These diagrams are multipartite graphs with four types of nodes: *or* gates, *and* gates, *basic* events and *intermediate* events.

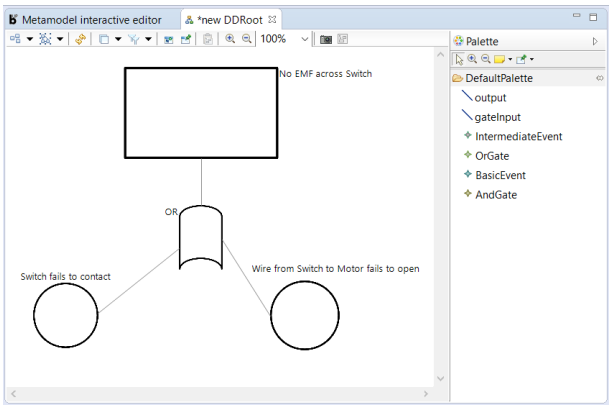


Fig. 25: An editor for fault-tree diagrams.

Building a multipartite graph-like DSL using our approach typically requires providing fragments that illustrate the properties of each node type and how they can be connected. In this case, we used the two fragments shown in Figure 26, which include three different types of objects: circles represent basic events, rectangles are intermediate events, and gates are labelled with their type. The first fragment uses 4 objects and 3 connectors, while the second one uses 7 and 6, respectively. None of the two fragments make use of spatial relationships, but just connections.

Figure 27 shows the textual representation extracted from the fragment to the right of Figure 26. Several references have been manually annotated with general (lines 4, 7, 16 and 19). Consequently, the derived meta-model generalizes these references by creating the abstract classes Event and Gate to correctly classify events and gates (see right of the figure). The name of the created abstract classes is automatically generated, being the common intersection of all subclasses names.

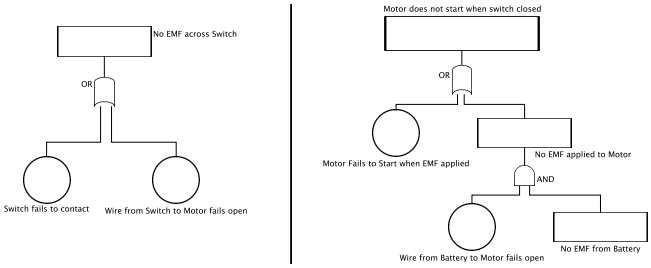


Fig. 26: Fragments used to define the DSL in Figure 25.

```
1  fragment yed_sketch {
2    "Motor does..." : IntermediateEvent {}
3    OrGate_1 : OrGate {
4      @general ref output = "Motor does..."
5    }
6    "Motor Fails..." : BasicEvent {
7      @general ref gateInput = OrGate_1
8    }
9    "No EMF applied..." : IntermediateEvent {
10     ref gateInput = OrGate_1
11   }
12   "Wire from..." : BasicEvent {
13     ref gateInput = AndGate_1
14   }
15   "No EMF from..." : IntermediateEvent {
16     @general ref gateInput = AndGate_1
17   }
18   AndGate_1 : AndGate {
19     @general ref output = "No EMF applied..."
20   }
21 }
```

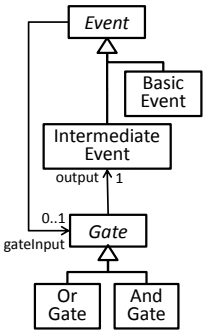


Fig. 27: Textual fragment (left). Derived meta-model (right).

8.2 Spatially defined languages

In this kind of languages, nodes are related via spatial relationships like adjacency, containment and overlapping. Typical languages belonging to this category are Venn diagrams and their variants [53], as well as grid-based languages such as board games like chess, checkers or Ludo [43].

As an example, Figure 28 shows the editor generated for the chess game. The editor permits creating the board, its cells and the pieces. We employed two fragments to generate this editor. The first fragment consisted of the board, which contained a grid of cells related to each other by adjacency and alignment spatial relationships. The second fragment illustrated the containment of white and black pieces inside cells.

8.3 Hybrid languages

Hybrid languages combine edge-based connections with spatial relationships. As an example of this kind of languages, Figure 29 shows an editor for use case diagrams. These diagrams employ containment and several types of nodes (actors, systems, packages and use cases) that can be connected with both directed and undirected edges. On the one hand, use cases can be contained in subsystems and packages. On the other, use cases can be connected between them with two types of directed edges, while actors can inherit from each other using directed

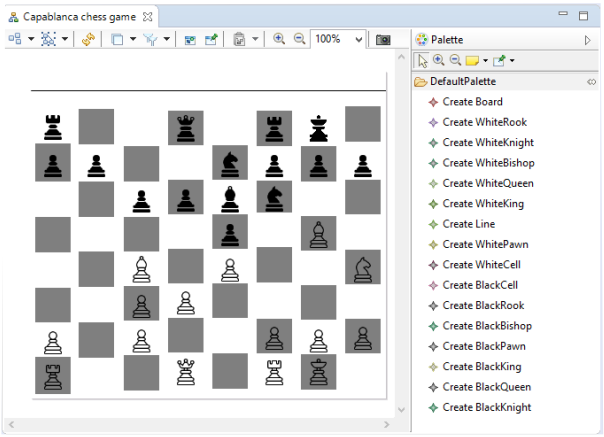


Fig. 28: A chess game editor.

edges, and be connected to use cases with undirected edges. In particular, the editor from Figure 29 was derived from 3 fragments including 5, 9 and 10 objects, which used 4 different types of objects. All fragments included some containment relationship between the objects: in the first fragment, one subsystem contained two use cases; in the second one, there was a subsystem with 5 use cases; and the last fragment included 3 packages containing several use cases.

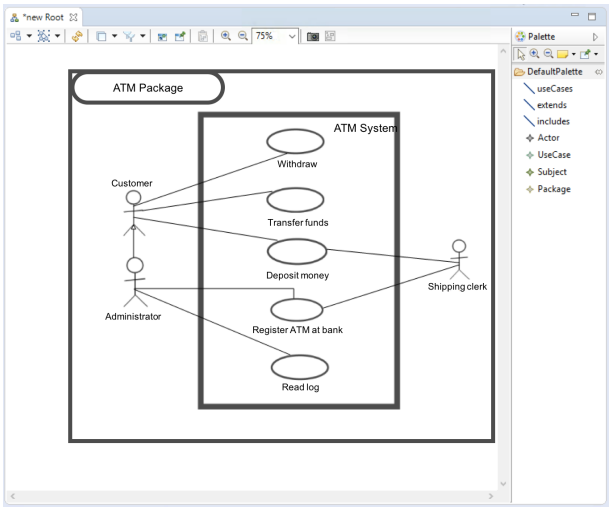


Fig. 29: An editor for use case diagrams.

Component diagrams are another example of hybrid language. They have been used to evaluate graphical DSL generation frameworks, like Eugenia [30]. This kind of diagram has plex features as components are interconnected via ports, and containment is relevant to indicate the subcomponents of a composed component. Figure 30 shows an editor for component diagrams, where there are components inside components, and some components are interconnected through their attached ports. This editor was inferred from only 2 fragments using 2 different object types. The fragments included 8 and 4 objects each, consisting in a number of components that exhibited an overlapping relationship with several ports connected between them.

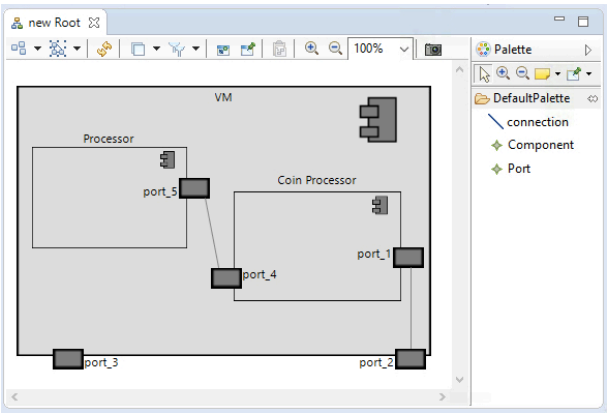


Fig. 30: An editor for component diagrams.

To build the editors for these languages, the provided fragments need to illustrate the allowed connections and spatial relationships. For the case of component diagrams, the containment relation between components is optional, as there may be top-level components and subcomponents. Hence, fragments reflecting both cases need to be provided.

8.4 Strengths and limitations of the approach

Next, we characterise the kinds of DSLs our approach supports. In general, as we have shown, our approach permits creating editors for the three kinds of languages identified in [8]: connection-based, spatially defined, and hybrid. However, we have to mention some limitations.

First, our approach supports DSLs where there is a one-to-one mapping between the abstract syntax and the concrete syntax. Hence, every class in the meta-model must be represented through an icon or an edge type. We do not support a combination of abstract syntax elements to be represented as a single icon, or having several concrete syntax representations for the same abstract syntax element. This leaves out languages like sequence diagrams, where the concrete and abstract syntax are very dissimilar. Likewise, our approach does not support variations on the concrete syntax elements (icons or edges) depending on attribute values, as it occurs for example in UML where associations can have decorations in case they are compositions, and class names are shown in italics when the class is abstract.

Edges can have decorators in the source and target ends but not in the middle, and we currently do not support edge labels.

Concerning spatial relationships, our algorithm identifies containment, adjacency, alignment and overlapping. However, adjacency only works for rectangular elements, as it is calculated based on the elements' bound-

ing box. This means adjacency cannot generally work, e.g., with triangles, and hence, complex adjacency-based languages like Nassi-Shneiderman diagrams or structograms [57] cannot be described with our approach. Moreover, our current target Sirius does not support overlapping. Instead, we give the option to substitute it by adjacency or containment. Hence, languages heavily relying on overlapping, like Venn and Euler diagrams [7, 51], cannot be handled. Regarding alignment, it is only reified on the presence of adjacency. While this is a reasonable heuristic for most engineering diagrams, it leaves out some pure spatially defined languages like the Braille language for the blind.

Finally, line-based diagrams, like those representing knots [44], cannot be recognized either. This is so as our approach is targeted to DSLs that include node types, and line-to-line relationships like overpassing or underpassing are not recognised.

Altogether, the main limitations of our approach come from being unable to handle spatial relationships over non-rectangular shapes, or very dissimilar concrete and abstract syntaxes. Anyhow, most languages in software engineering are hybrid (e.g., like those in Figures 29 and 30), do not make use of complex spatial relationships, and relations are most frequently depicted as edges (maybe via ports) and containment. The running example and the DSLs in this section illustrate the kind of languages our approach is suitable for.

9 Related work

In this section, we review related approaches to the definition of DSL requirements (Section 9.1) and the generation of flexible modelling tools (Section 9.2), and perform a feature-based comparison of the main tools for the flexible creation of graphical DSLs (Section 9.3).

9.1 DSL requirements

According to [33,39], domain analysis techniques are still missing for DSLs, as this phase is most frequently done in an informal way. Our work uses drawings built with diagramming tools to represent DSL requirements. Other ways to represent requirements include feature diagrams [23] or notations inspired by mind-maps [42]. While these approaches focus on representing and classifying desired DSL features, our example-based approach relies on concrete examples of language use, promoting a more direct involvement of domain experts.

In some sense, our approach is conceptually similar to test-driven development approaches [6]. This is so as, in test-driven development, the software requirements are expressed as test cases, and the software is evolved to make it pass the tests; while in our approach, the example models can be seen as DSL requirements, and we provide an automated process that evolves the current version of the DSL meta-model so that it accepts the examples.

In [1], domain analysis is application-driven and DSLs are produced as output artefacts when developing a software framework, while in [49], the Question-Options-Criteria guides the design decisions involved when creating UML-based DSLs. However, none of these two works propose a concrete notation to represent DSL requirements.

9.2 Flexible modelling

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 [38,46,56], or as a means to promote an active role of domain experts in DSL development [9,14,59], as we advocate in this paper.

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 [2,4,5,24,27,55] and model refactorings [13]. Some approaches, like [2], rely on mappings specified manually over examples with a graphical concrete syntax. However, the use of example-based techniques is not so common to describe graphical modelling environments. The closest work to ours is the MLCBD framework [9], 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 deduced meta-model supports modelling concepts like abstract classes, inheritance, compositions and attributes, which are not found in [9].

The approach in [31,59], called Muddles, uses yED to draw examples of the DSL. Types are assigned to elements based on *type* annotations, and functions can be defined to check for shape overlapping, colour or proximity. *Type* annotations are placed on every node in the drawing, and can indicate subtyping relations. All modelling is performed within yED, and no dedicated modelling environment is explicitly generated. The rationale of the approach is to enable early use of model management languages over the drawings. For this purpose, Muddles permits the use of the Epsilon model

management languages to query and manipulate the yED models directly. This is possible because the Epsilon languages communicate with the modelling platform through a connectivity layer. Muddles does not expose a meta-model to the user, but instead, a meta-model is created in-memory from the type annotations. This meta-model deliberately leaves out elements like cardinalities or composition references.

The Free Modeling Editor (FME) presented in [14] permits developing DSLs starting either from example models or from meta-model concepts. The proposal is based on the Openflexo tool, which supports the concurrent development of both models and meta-models, and has importers for models stored in PowerPoint and Word formats. The approach has been successfully applied to an industrial project [14].

Some tools for DSL development generate an external modelling tool different from the one used to define the DSL. For instance, EuGENia Live [45] is a web tool for designing graphical DSLs. It supports on-the-fly meta-model editing while the user is building a sample model and its concrete syntax. From this definition, the tool exports an Ecore meta-model enriched with concrete syntax annotations, which can be used to generate an Eclipse GMF-based environment.

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 [46] provides an API to enable sketch-based editing on Eclipse; Calico [38] is a sketching tool for electronic whiteboards, where the sketched elements can be scrapped and reused in other parts of the diagrams; and FlexiSketch [56] allows creating sketches, and to manually lift the created shapes and connections to the meta-model level. However, FlexiSketch does not support meta-modelling features like named attributes, class inheritance, abstract classes or different association types (e.g., compositions).

Finally, although not specific for DSL creation, there is a trend in recent modelling tools to promote flexibility by relaxing the conformance relationship in early phases of modelling, while enforcing strictness in later phases [18,50]. These tools benefit from the flexibility of JavaScript as the underlying implementation language.

Altogether, bottom-up approaches to DSL construction (FME, MLCBD, metaBup) foster an active participation of domain experts, who provide examples that drive the DSL construction process, including the derivation of the meta-model. However, for large meta-models (e.g., found in large standards like UML or BPMN) a combination with top-down meta-model design would be more adequate, to organise and architect the different parts of the meta-model. Regarding DSL evolution, the relaxed model/meta-model conformity provided by flexible modelling approaches might be useful to cope with non-conforming models. Moreover, the incorporation of new requirements to an existing DSL might be simplified with bottom-up approaches, if those requirements can be expressed as a new example.

9.3 Comparison of flexible tools to build graphical DSLs

Table 4 compares our approach metaBup with the most prominent tools for the flexible creation of graphical DSLs, namely, EuGENia Live [45], FlexiSketch [56], FME [14], MLCBD [9] and Muddles [31,59]. In the following, we comment on the differences between the features of these tools.

9.3.1 Approach First, we compare the overall approach the tools implement. metaBup and EuGENia Live rely on informal drawing tools to specify examples, and then generate an external modelling tool that mimics the exemplified graphical syntax. In particular, metaBup generates a Sirius-based modelling editor, and EuGENia Live generates a GMF-based one. In contrast, MLCBD and Muddles also start from informal drawings, but then modelling is performed in the same drawing tools (i.e., no external modelling tool is generated). The remaining cases (FlexiSketch and FME) do not rely on external drawing tools or graphical modelling frameworks, but they are themselves flexible self-contained modelling environments. FME, in addition, can also import drawings from some external tools, but the meta-model needs to be manually created.

9.3.2 Process Second, we compare the process followed to define the DSL. metaBup implements a bottom-up approach, where the provided examples are used to automatically deduce a meta-model making explicit the syntactic rules of the DSL. FlexiSketch and Muddles also deduce a meta-model, however their use is internal and does not get exposed to the user. Instead, MLCBD does not infer or need an explicit meta-model, while EuGENia Live and FME require creating a meta-model manually.

9.3.3 DSL examples Table 4 also compares how the examples are created in the different tools. Some of them use existing popular drawing tools like yED, Dia or the Microsoft Office suite. The advantage is that domain experts might find familiar some of them. Instead, EuGENia Live and FlexiSketch require using the modelling tool itself to draft the examples.

Like our approach, Muddles relies on yED drawings, where each node should be annotated with its type. This is done using *type* extension fields. In contrast, our type names are taken from the icon used to create each node. This is less demanding for the user, who does not need to explicitly annotate each node.

9.3.4 DSL meta-model As abovementioned, metaBup, FME, Muddles and FlexiSketch allow building meta-models from the examples, although the latter two do

Table 4: Flexible approaches for DSL creation

	Approach	**Process**	**DSL examples**	**DSL meta-model**	**Modelling environment**	**Advanced recognition**
metaBup	informal drawings + editor generation	examples + meta-model derivation	yED, Dia	automatic (rich meta-modelling features)	generated (Eclipse)	spatial relations
EuGENia Live [45]	informal drawings + editor generation	examples + meta-model	inside tool	manual	generated (Eclipse)	-
FlexiSketch [56]	flexible tool (sketches+models)	examples + meta-model derivation	inside tool	automatic (basic meta-modelling features)	inside tool	sketching
FME [14]	flexible tool (examples+models)	examples + meta-model	inside tool, Office	manual	inside tool	-
MLCBD [9]	informal drawings + informal editor	examples	Ms Visio	-	Ms Visio	-
Muddles [31, 59]	informal drawings + informal editor	examples	yED	automatic (type annotations)	yED	spatial relations

not expose the meta-model to the user. Muddles relies on type annotations to derive the meta-model, and allows the specification of type hierarchies. While it can recognise spatial relationships programmatically, they are not reified as meta-model references, which makes model management cumbersome. FlexiSketch permits assigning a type name to the elements created in a sketch. However, the deduced meta-models do not support regular modelling concepts like inheritance, abstract classes, compositions or constraints. Nodes can be assigned several untyped labels to emulate attribute values, but they have no name and their type-checking w.r.t. basic data types cannot be performed. FME requires manual creation of the meta-model, but supports features like abstract classes, inheritance, cardinalities and full-fledged attributes.

Similar to FME, metaBup supports conceptual modelling facilities like composition, attributes, cardinalities and inheritance. In contrast to FME, the meta-model is automatically deduced from the examples. Moreover, metaBup incorporates a catalogue of refactorings that can be used to improve the meta-model quality, and an assistant able to recommend those refactorings.

9.3.5 Modelling environment Regarding the modelling environment obtained from the DSL definition, metaBup and EuGENia Live generate external modelling environments for Eclipse, MLCBD and Muddles enable the tools they use to draw examples as modelling environments, and FlexiSketch and FME support modelling inside them.

We believe that creating a dedicated meta-model and modelling environment atop a meta-modelling framework has several benefits. First, a modelling environment provides customised forms to create and edit objects of the different meta-model types, with appropriate fields for the object attributes and facilities for their type checking. Instead, attribute values must be specified via tags in drawing tools like Visio and yED, and there is no conformance or type checking. Second, the created models can be manipulated using standard model management languages for model transformation or code generation.

9.3.6 Advanced recognition Some of the analysed tools can recognise advanced graphical aspects (i.e., beyond the identification of nodes and edges) in the provided examples. Muddles identifies spatial relationships, like proximity or overlapping. Similar to our approach, overlapping is recognised based on the bounding box of elements, so overlapping of shapes like triangles or circles is only approximated. Spatial relationships are not reified in its internal meta-model, and they are not enforced when modelling. FlexiSketch is the only analysed tool that supports sketching, but it does not provide spatial relationship recognition. Among the approaches that generate a dedicated environment, metaBup is the only one able to identify spatial relationships between elements, and enforce them in the generated modelling environments.

9.3.7 Comparison based on the running example We have used the tools from Table 4 that are publicly available, to develop the DSL of our running example. In particular, we were able to use FlexiSketch and FME. Muddles is also available, but as modelling occurs within yED, the resulting editor would look like our fragments (see, e.g., Figure 14).

Figure 31 shows the running example DSL built with FlexiSketch. The tool works on mobile devices and Android tablets. The node icons can be both images and hand-made sketches, though we opted for the latter to avoid transparency issues. The first time a node is created, it can be assigned a type name. Nodes can be connected using edges with different styles. While we have emulated containment of devices inside the home, overlapping of the modem and the house, and adjacency of ports and the router, these spatial relationships do not have any meaning because the tool does not interpret them. Instead, it is necessary to provide explicit edges. We could not provide information of edge cardinalities, but edge labels are supported. Attributes can be emu-

lated through plain text annotations with no name and no declared type.

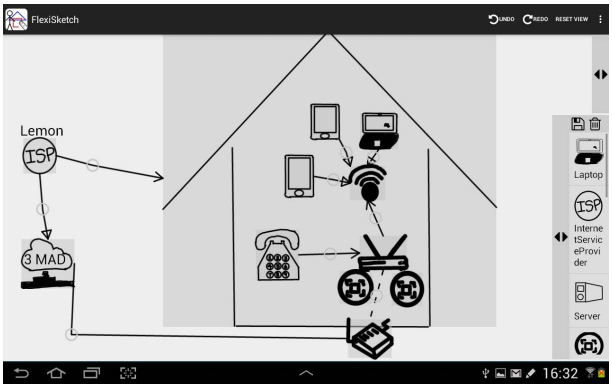


Fig. 31: Running example DSL in FlexiSketch.

Figure 32 shows how the running example DSL looks like in FME. This tool permits importing example drawings from PowerPoint files, or directly drawing example models in the tool. Then, the user can select manually which shapes in the models should become concepts of the meta-model. This step is similar in FlexiSketch. Concepts can be enriched with full-fledged attributes of a set of predefined data types, be declared abstract, or inherit from other concepts. Edges in examples can also be lifted to meta-model associations, and define cardinalities. Similar to FlexiSketch, the tool does neither support nor recognize spatial relationships, which need to be specified in models using edges. Hence, although Figure 32 shows devices inside the home, a modem overlapping the house, and ports adjacent to the router, these relationships do not have any impact on semantics.

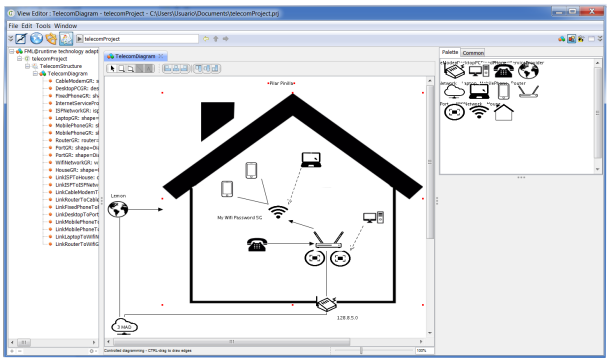


Fig. 32: Running example DSL in FME.

In summary, FME is able to import examples from PowerPoint, similar to our approach. Both FME and FlexiSketch support bottom-up meta-modelling, though it is not automatic, but the modelling expert must select manually the model elements to be lifted to the meta-model, and configure attributes and cardinalities by hand. Instead, our approach derives a meta-model automatically upon the provision of new examples. Hence, the burden of the modelling expert is lower in our case. Moreover, deriving the meta-model automatically prevents errors derived from forgetting assigning types to model elements. According to [58], this is an issue in flexible modelling approaches that manually construct the meta-model. None of these two tools generate an explicit meta-model, and the resulting editor is embedded in the tool itself. In contrast, we produce an explicit meta-model and a separated modelling environment. This results in a customized environment for the end user, which is not mixed with meta-modelling functionalities. As our environment is based on EMF, it can be combined with model management tools of the rich EMF ecosystem. Finally, neither FlexiSketch nor FME support spatial relationships, a salient feature of our approach. Instead, these need to be emulated using edges.

9.4 Summary

Altogether, our approach is novel as it enables the creation of graphical DSL editors based on drawings produced by domain experts, automatically generating a meta-model and a dedicated modelling environment. With respect to existing flexible modelling tools (see Table 4), it has advantages like recognition of spatial relationships, reification of those in the meta-model, and explicit generation of a meta-model and a custom graphical 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.

10 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 examples of the DSL built with diagramming tools, and our system derives 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). We have conducted a user study that shows positive results and encourages further research on this approach to DSL creation.

In the future, we plan to facilitate the validation of the final editor by the domain experts by integrating our

mmXtens language [36], which is able to generate example models satisfying certain properties of interest 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 [32], where manual changes are described as a program that is reapplied when re-generation occurs. Similarly, we also plan to provide support for meta-model/model co-evolution. Another interesting aspect is to integrate mechanisms for assessing the quality of the created DSL within our process, in the style of [22,41,47], and improve the tool capabilities regarding concrete syntax. A particular challenging aspect is the inference of conditional styles. We will also improve the bottom-up meta-model construction process by providing support for enumerations, and investigating possible effects of fragment ordering.

Regarding the provision of fragments, we currently do not maintain traceability between the graphical and textual fragments. If the domain expert changes a graphical fragment, this should be imported again into the system, and it would be considered as a new fragment. Hence, we plan to add traceability support, as well as the possibility to update or roll back some previously introduced example.

Acknowledgements

Work funded by the Spanish Ministry of Economy and Competitivity (TIN2014-52129-R), and the R&D programme of the Madrid Region (S2013/ICE-3006).

References

1. X. Amatriain and P. Arumí. Frameworks generate domain-specific languages: A case study in the multimedia domain. *IEEE Trans. Software Eng.*, 37(4):544–558, 2011.
2. I. Avazpour, J. Grundy, and L. Grunske. Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations. *J. Vis. Lang. Comput.*, 28:195–211, 2015.
3. 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.
4. I. Baki and H. A. Sahraoui. Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Trans. Softw. Eng. Methodol.*, 25(3):20, 2016.
5. Z. Balogh and D. Varró. Model transformation by example using inductive logic programming. *Software and System Modeling*, 8(3):347–364, 2009.
6. K. Beck. *Test Driven Development: by Example.* Addison-Wesley Professional, 2003.
7. P. Bottoni and A. Fish. Coloured euler diagrams: A tool for visualizing dynamic systems and structured information. In *Proc. Diagrams*, volume 6170 of *Lecture Notes in Computer Science*, pages 39–53. Springer, 2010.
8. P. Bottoni and A. Grau. A suite of metamodels as a basis for a classification of visual languages. In *VL/HCC*, pages 83–90. IEEE Computer Society, 2004.
9. 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.
10. 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.
11. J. Feder. Plex languages. *Inf. Sci.*, 3(3):225–241, 1971.
12. 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.
13. A. Ghannem, G. El-Boussaidi, and M. Kessentini. Model refactoring using examples: a search-based approach. *Journal of Software: Evolution and Process*, 26(7):692–713, 2014.
14. F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, and C. Guychard. Using free modeling as an agile method for developing domain specific modeling languages. In *MoDELS*, pages 24–34. ACM, 2016.
15. Graphiti. `https://eclipse.org/graphiti/`.
16. R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, 2009.
17. J. C. Grundy, J. G. Hosking, K. N. Li, N. M. Ali, J. Huh, and R. L. Li. Generating domain-specific visual language tools from abstract visual specifications. *IEEE Trans. Software Eng.*, 39(4):487–515, 2013.
18. N. Hili. A metamodeling framework for promoting flexibility and creativity over strict model conformance. In *FlexMDE @ MoDELS*, volume 1694 of *CEUR Workshop Proceedings*, pages 2–11. CEUR-WS.org, 2016.
19. 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.
20. A. Jiménez-Pastor, A. Garmendia, and J. de Lara. Scalable model exploration for model-driven engineering. *Journal of Systems and Software*, 132:204–225, 2017.
21. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comp. Programming*, 72(1):31–39, 2008.
22. G. Kahraman and S. Bilgen. A framework for qualitative assessment of domain-specific languages. *Software and System Modeling*, 14(4):1505–1526, 2015.
23. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.
24. 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.
25. S. Kelly and R. Pohjonen. Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29, 2009.

26. S. Kelly and J. Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008.
27. M. Kessentini, H. A. Sahraoui, M. Boukadoum, and O. Benomar. Search-based model transformation by example. *Software and System Modeling*, 11(2):209–226, 2012.
28. A. J. Ko, R. Abraham, L. Beckwith, A. F. Blackwell, M. M. Burnett, M. Erwig, C. Scaffidi, J. Lawrance, H. Lieberman, B. A. Myers, M. B. Rosson, G. Rothermel, M. Shaw, and S. Wiedenbeck. The state of the art in end-user software engineering. *ACM Comput. Surv.*, 43(3):21, 2011.
29. A. J. Ko, T. D. LaToza, and M. M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1):110–141, 2015.
30. D. S. Kolovos, A. García-Domínguez, L. M. Rose, and R. F. Paige. Eugenia: towards disciplined and automated development of gmf-based graphical model editors. *Software and System Modeling*, 16(1):229–255, 2017.
31. D. S. Kolovos, N. D. Matragkas, H. H. Rodriguez, and R. F. Paige. Programmatic muddle management. In *XM@MoDELS*, volume 1089 of *CEUR Workshop Proceedings*, pages 2–10. CEUR-WS.org, 2013.
32. 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.
33. T. Kosar, S. Bohra, and M. Mernik. Domain-specific languages: A systematic mapping study. *Information & Software Technology*, 71:77–91, 2016.
34. 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.
35. J. J. López-Fernández, A. Garmendia, E. Guerra, and J. de Lara. Example-based generation of graphical modelling environments. In *ECMFA*, volume 9764 of *LNCS*, pages 101–117. Springer, 2016.
36. 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.
37. J. J. López-Fernández, E. Guerra, and J. de Lara. Combining unit and specification-based testing for meta-model validation and verification. *Inf. Syst.*, 62:104–135, 2016.
38. 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.
39. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
40. Microsoft. `https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx`, 2017.
41. D. L. Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Software Eng.*, 35(6):756–779, 2009.
42. A. Pescador and J. de Lara. Dsl-maps: from requirements to design of domain-specific languages. In *ASE*, pages 438–443. ACM, 2016.
43. A. Rensink, A. Dotor, C. Ermel, S. Jurack, O. Kniemeyer, J. de Lara, S. Maier, T. Staijen, and A. Zündorf. Ludo: A case study for graph transformation tools. In *Proc. AGTIVE*, volume 5088 of *Lecture Notes in Computer Science*, pages 493–513. Springer, 2007.
44. M. D. Rosa, A. Fish, V. Fuccella, R. Saleh, S. Swartwood, and G. Costagliola. A toolkit for knot diagram sketching, encoding and re-generation. In *Proc DMS*, pages 16–25. KSI Research Inc. / Knowledge Systems Institute Graduate School, 2016.
45. L. M. Rose, D. S. Kolovos, and R. F. Paige. Eugenia live: A flexible graphical modelling tool. In *XM @ MoDELS*, pages 15–20. ACM, 2012.
46. U. B. Sangiorgi and S. D. Barbosa. SKETCH: Modeling using freehand drawing in eclipse graphical editors. In *FlexiTools @ ICSE*, 2010.
47. O. Semeráth, A. Barta, A. Horváth, Z. Szatmári, and D. Varró. Formal validation of domain-specific languages with derived features and well-formedness constraints. *Software and System Modeling*, In press, 2016.
48. Sirius. `https://eclipse.org/sirius/`.
49. S. Sobernig, B. Hoisl, and M. Strembeck. Extracting reusable design decisions for uml-based domain-specific languages: A multi-method study. *Journal of Systems and Software*, 113:140–172, 2016.
50. J. Sottet and N. Biri. JSMF: a javascript flexible modelling framework. In *FlexMDE @ MoDELS*, volume 1694 of *CEUR Workshop Proceedings*, pages 42–51. CEUR-WS.org, 2016.
51. G. Stapleton, J. Howse, J. Taylor, and S. J. Thompson. The expressiveness of spider diagrams. *J. Log. Comput.*, 14(6):857–880, 2004.
52. G. Stapleton, M. Jamnik, and A. Shimojima. What makes an effective representation of information: A formal account of observational advantages. *Journal of Logic, Language and Information*, 26(2):143–177, 2017.
53. G. Stapleton, S. J. Thompson, A. Fish, J. Howse, and J. Taylor. A new language for the visualization of logic and reasoning. In *Proc. DMS*, pages 287–292. Knowledge Systems Institute, 2005.
54. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley Professional, NJ, 2008.
55. Y. Sun, J. Gray, and J. White. A demonstration-based model transformation approach to automate model scalability. *Software and System Modeling*, 14(3):1245–1271, 2015.
56. 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.
57. C. Yoder and M. Schrag. Nassi-Shneiderman charts: an alternative to flowcharts for design. In *ACM SIGSOFT/BIGMETRICS Software and Assurance Workshop*, pages 386–393, 1978.
58. A. Zolotas, R. Clariso, N. Matragkas, D. S. Kolovos, and R. F. Paige. Constraint programming for type inference in flexible model-driven engineering. *Computer Languages, Systems & Structures*, (to appear):–, 2017.
59. 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.

Appendix 1

This appendix contains the following documents provided to the participants in our evaluation: formulation of the expected DSL requirements, instructions of the experiment, and questionnaires. Questionnaire 1 was answered by all the participants, in order to convey their opinions on the first version of the generated modelling tool. Questionnaire 2 was only answered by the participants that opted for generating a second version of the modelling tool. Mandatory questions are marked with an asterisk.

Formulation of DSL requirements:

Next, we describe a domain for which we want to create a graphical modelling language. For this purpose, we only need you to provide some drawings with the appearance you would like to have in your models.

Description: *We want to model domestic networks in which a Service Provider (ISP) provides Internet access to its clients by means of local networks. Each client hosts a domestic net, connected to a unique local network via a modem. Each client can also have a router (which can be enabled for WiFi connections) with a number of ports to which different devices can be connected. The router has access to Internet through the modem.*

Devices are connected to the net in a different way, depending on whether they are wireless or not. Printers, desktop computers, servers and fixed phones are connected through the router. They need to be plugged to one of the router ports, but the fixed phone which is directly connected to the router. On the other hand, wireless devices (laptops, Smart TVs and smartphones) can only access Internet through a WiFi network, which will be the one connected to the router directly.

Instructions:

You can provide as many drawings as you deem necessary, each one illustrating domain examples according to your own criterion. They do not mandatorily have to be complete examples, but they can focus on specific aspects of the problem domain. For instance, you can make a diagram with just some local networks connected to Internet, another with the configuration of a WiFi network, etc.

The drawings do not need to include unnecessary elements for the aspect you want to model, while you should make as many drawings as necessary to illustrate all aspects of the domain. Each element in the palette should be used at least once in some drawing.

When making the drawings, please remind that the following properties are meaningful. (i) If you want to establish several kinds of connections between two objects, you should use different line styles. (ii) The relative position of objects is meaningful. This means that drawings can make use of containment, adjacency and overlapping of objects. (iii) You can add information to objects by attaching them labels with the following format: $\langle field - name \rangle = \langle field - value \rangle$.

Questionnaire 1:

Q1 *Indicate your age * :* _____

Q2 *Indicate your gender * :*
- ○ *Male*
- ○ *Female*

Q3 *Indicate your current workplace * :*
- ○ *University*
- ○ *Information technology company*
- ○ *Different sector*
- ○ *Unemployed*

Q4 *Does the synthesized modelling environment meet the graphical syntax you envisioned when providing the examples? **
Not at all ○1 ○2 ○3 ○4 ○5 Very much

Q5 *Which aspects of the graphical language are not correctly captured by the modelling environment?* _____

Q6 *Which aspects of the graphical language are best captured by the modelling environment?* _____

Q7 *How easy is it to use the modelling environment? **
Very difficult ○1 ○2 ○3 ○4 ○5 Very easy

Q8 *Which aspects of the environment would you improve?* _____

Q9 *Which aspects of the environment do you like the most?* _____

Q10 *Which is your higher level of expertise with modelling? **
- ○ *I have developed both meta-models and graphical domain-specific languages.*
- ○ *I have developed meta-models, but not graphical domain-specific languages.*
- ○ *I have used domain-specific languages like UML or BPMN.*
- ○ *I have never used or developed models or meta-models.*

Answer the following questions only if you selected one of the first two choices in the previous question (Q10).

Q11 *Is the meta-model generated from the examples similar to the one you would have built by hand? **
Not at all ○1 ○2 ○3 ○4 ○5 Very much

Q12 *Which aspects does the meta-model not capture correctly? **_____

Q13 *Which approach would you prefer to build the meta-model? **
- ○ *I would prefer designing the meta-model myself.*
- ○ *I would prefer using examples, and then being able to modify the meta-model manually.*
- ○ *I would prefer using examples, and I do not think necessary to modify the meta-model manually.*

Questionnaire 2:

Q1 *Has the editor quality been improved with respect to the first iteration?*
Not at all ○*1* ○*2* ○*3* ○*4* ○*5* *Very much*

Q2 *Which aspects of the language are still not reflected in the editor?*
- ○ *None*
- ○ *Other:* _____

Appendix 2

Table 5 shows the changes made by the modelling expert on the fragments provided by the participants in our evaluation (cf. Section 7). The columns contain the participant identifiers, the number of fragments, the editing actions, and the time employed to commit them in seconds.

Table 5: Manual changes made by the modelling expert to the fragments provided in the evaluation, and time taken

Part.	#Fr.	Edits made by modelling expert	Time (s)
1	1	– Rename references	19
2	1	– Rename references	27
3	1	– Rename references – Delete references ignoring the invisible bounding box of objects (see Figure 24) – Fix inconsistent spatial relations – Change the type of an attribute from String to Integer; this attribute had been assigned the value 'n' to indicate a number in the fragment	44
4	2	– Rename references – Rename auto-generated superclass	49
5	2	– Rename references	12
6	2	– Rename references	70
7	2	– Rename references – Adjust containment spatial relations (for CableModem)	78
8	3	– Rename references – Move object misplaced in the graphical sketch	66
9	3	– Rename references – Delete references ignoring the invisible bounding box of objects (see Figure 24) – Move object misplaced in the graphical sketch – Adjust containment spatial relations (for CableModem and Router)	128
10	4	– Rename references	77
11	6	– Rename references – Adjust containment spatial relations	224