Zurich Open Repository and Archive
University of Zurich
University Library
Strickhofstrasse 39
CH-8057 Zurich
www.zora.uzh.ch

Year: 2013

Semi-automatic Generation of Metamodels from Model Sketches

Wüest, Dustin ; Seyff, Norbert ; Glinz, Martin

DOI: https://doi.org/10.1109/ASE.2013.6693130

Posted at the Zurich Open Repository and Archive, University of Zurich
ZORA URL: https://doi.org/10.5167/uzh-88828
Conference or Workshop Item
Published Version

Originally published at:
Wüest, Dustin; Seyff, Norbert; Glinz, Martin (2013). Semi-automatic Generation of Metamodels from Model Sketches. In: IEEE/ACM International Conference on Automated Software Engineering, Silicon Valley, California, 11 November 2013 - 15 November 2013. IEEE, 664-669.
DOI: https://doi.org/10.1109/ASE.2013.6693130

Semi-automatic Generation of Metamodels from Model Sketches

Dustin Wüest, Norbert Seyff, Martin Glinz
Department of Informatics, University of Zurich, Switzerland
{wueest,seyff,glinz}@ifi.uzh.ch

***Abstract*—Traditionally, metamodeling is an upfront activity performed by experts for defining modeling languages. Modeling tools then typically restrict modelers to using only constructs defined in the metamodel. This is inappropriate when users want to sketch graphical models without any restrictions and only later assign meanings to the sketched elements. Upfront metamodeling also complicates the creation of domain-specific languages, as it requires experts with both domain and metamodeling expertise.**

In this paper we present a new approach that supports modelers in creating metamodels for diagrams they have sketched or are currently sketching. Metamodels are defined in a semi-automatic, interactive way by annotating diagram elements and automated model analysis. Our approach requires no metamodeling expertise and supports the co-evolution of models and metamodels.

***Index Terms*—Sketch, model, metamodel, inference, semi-automated, end-user.**

I. Introduction

With the advent of model-driven engineering (MDE), models have become the main artifacts in a tool-supported, model-centric development process [4]. Such an approach requires models to be machine-processable and transformable. Consequently, the corresponding modeling languages need to be defined precisely. Graphical modeling languages, on which we focus in this paper, are typically defined by a metamodel [2].

The standard way of using a modeling language is to define the language first, i.e., experts must create a metamodel for a new language before modelers can use the language for creating actual models [18]. This allows the creation of powerful analysis and transformation tools required for MDE.

However, in the early phase of development, for eliciting, creating and sketching initial ideas, engineers want and need freedom in choosing notations adapted to their needs – be it in the form of domain-specific languages (DSLs), by back-of-an-envelope style sketches, or both. Standard modeling languages such as UML are not well suited for that purpose. Instead, we need languages that can be flexibly defined and used in a way that they are well adapted for the specific problem at hand.

The traditional paradigm of upfront metamodeling breaks down here: Modelers want the flexibility to draw model elements regardless whether or not a pre-defined metamodel provides such elements [21]. DSL designers want to draw sample models in a new DSL with full tool support before formally defining model elements in a metamodel.

At the same time, however, there is still a need for evolving such flexibly created models into a form that allows formal analysis and transformations. That means that metamodels must be created at some point. Metamodeling tools such as MetaEdit+ [27] and MetaBuilder [12] provide some relief by making the task of formally defining a DSL easier and faster, but they still require upfront metamodeling. Today, modelers who need flexible modeling capabilities frequently use whiteboards for sketching [5][19]. This is done at the expense of later re-creating the sketched models manually in a more formal modeling language in order to feed them into an MDE chain.

For really solving the flexible modeling problem, we need to *interleave modeling and metamodeling* activities and a tool that supports the *co-evolution of models and metamodels*. Combining sketching and metamodeling in a single tool is an approach not well studied so far. In our own previous work, we have developed the FlexiSketch approach [30][31] which allows free interleaving of modeling and metamodeling tasks.

In this paper, we present how FlexiSketch step-wise and semi-automatically generates metamodels for model sketches, by inferring metamodel clues from existing model fragments and interactively eliciting missing metamodel information. Thereby, FlexiSketch enables modelers with no prior metamodeling expertise to annotate sketched model elements with meanings and eventually produces a fitting metamodel.

The remainder of this paper is structured as follows. Section II provides the objectives of our research and information about FlexiSketch. Section III discusses FlexiSketch's metamodeling capabilities. Section IV outlines first evaluation results. Section V presents related work. Section VI concludes, discusses limitations and future work.

II. Modeling Languages and FlexiSketch

A. Focus and Objectives of our Work

We are interested in generating definitions of concrete and abstract syntax from a set of existing model sketches. Our goal is enabling engineers to create a language syntax definition for their early model sketches, such that

- these sketches can be formalized and re-used during the development process of a software project,
- engineers don't need metamodeling expertise,
- the tasks of modeling and metamodeling can be interleaved.

DOI 10.1109/ASE.2013.6693130
© 2013 IEEE

Proceedings ASE 2013, Palo Alto, USA
New Ideas Track

We restrict the scope of our work to graphical, node-and-edge style models. Typical examples for such diagrams are class diagrams, component diagrams or activity diagrams in UML. Also, graphical DSLs typically fall into this category. This scope allows us to restrict the metamodel elements and structure we need to consider, omitting complicated structures such as deep inheritance trees that are hard to understand even for experts.

We focus on *collecting* metamodel information, both automatically by inference from existing model sketches and interactively in a tool-guided dialog with the engineer. Producing metamodels compatible with those of existing modeling tools is beyond the scope of our current work. However, we intend to generate metamodels that are sufficiently formal so that they can be transformed into a format understood by a commercial modeling tool.

B. FlexiSketch in a Nutshell

Our approach has been implemented prototypically in our FlexiSketch tool [31]. It is available for Android OS tablet devices and supports lightweight and flexible modeling. Having a mobile tool allows to use it in-situ in various contexts.

On start-up, FlexiSketch tries to mimic a whiteboard. Most of the screen is empty, inviting users to start sketching. User drawings are converted into elements that can be manipulated (e.g., moved, scaled, or deleted). FlexiSketch differentiates between symbols (nodes) and links (edges). Nodes may be drawn or consist of imported images. The tool allows assigning types to sketched elements by annotating them. These annotations provide the basic structure of a metamodel. Graphical representations of user-defined types appear in a *type library* on the right side of the screen. From there, users can create copies of their elements by dragging them onto the drawing canvas. Thus, the type library is a container for the user-defined concrete syntax. A sketch recognition algorithm processes the user-drawn symbols. If the user draws a symbol that looks similar to one from the type library, the tool asks in a small popup window whether it is the same symbol type. A more detailed description of FlexiSketch is given in [31].

Figure 1 shows a screenshot of the tool, showing a model sketch. The top right symbol is currently selected (indicated by a blue background and the visible context menu icons).

C. Using FlexiSketch – A Scenario

Engineers can use FlexiSketch to freely sketch their ideas as node-and-edge type models without any well-formedness constraints. If they decide to keep the resulting artifacts, FlexiSketch provides them with an easy, user-friendly way to add a metamodel to their sketches by annotating elements and answering questions asked by FlexiSketch's tool wizard. Once all the metamodel information has been collected, the model sketches and their metamodel(s) can be exported into an XML file. This file can then be transformed such that the models can be imported into other modeling tools, thus supporting an MDE approach. This encourages engineers to include their early sketches systematically into the software engineering process and avoids costly and risky re-modeling of information originally documented in sketches.

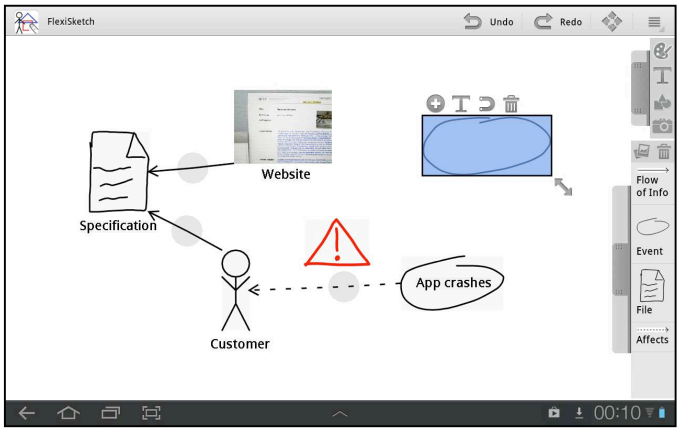


Fig. 1. The FlexiSketch tool showing a user's sketch.

III. Metamodeling in FlexiSketch

In this Section, we give an overview on metamodel fundamentals in FlexiSketch, and then explain how we build metamodels based on inference and tool guidance. We also show how we minimize the versioning problem when associating metamodels with existing and new sketches.

A. The Metamodel Structure in FlexiSketch

In FlexiSketch, the user can create symbols, links, and annotations as elements on the drawing canvas. Symbols and links are *TypedElements*, i.e., the user can define types for these elements. This creates a new *SymbolType* or *LinkType* class in the metamodel. Annotations are used to add informal notes to the sketched models. Furthermore, the meta-metamodel supports *Attributes* and *Containments*. Attributes can be used to add fields with type-value pairs to a symbol or link type. A containment gets created when a symbol is part of another symbol, i.e. it is drawn inside another symbol. The containment then stores the types of the symbols together with cardinalities defining how many symbols of a particular type may be contained in the symbol of the other type. Attributes and containments are not yet supported in the tool.

FlexiSketch does not store cardinalities directly for link types. Instead, it stores them for *ConnectionTypes*, which is a more flexible solution. While a link type is defined by just the type of the link itself, we uniquely identify a connection type as combination of the type of the link and the types of the two connected symbols. If the link is directed, we have a *start symbol* and an *end symbol*. A link type can have several connection types, i.e., when the same link type is used to connect different types of symbols. For example, a link type R may be used in one case to connect a symbol of type A with a symbol of type B, and in another case to connect a symbol of type A with a symbol of type C. Accordingly, the tool generates two connection types, one for $R(A, B)$ and one for $R(A, C)$. The connection type for $R(A, B)$ defines that R points from a symbol of type A to a symbol of type B. The cardinalities define (i) to how many type B symbols a single type A symbol may have outgoing links of type R, and (ii) from how many type A symbols a single type B symbol may have incoming type R links.

B. Recognizing Elements on the Drawing Canvas

For the automated model analysis, we assume that the various elements in a model are already categorized into nodes and edges. This categorization is directly tied to how model sketching works in the tool. Whenever the user starts drawing and then stops for a certain amount of time, that drawing is converted into a distinct symbol which is always a node. Links (edges) can only be created by connecting two previously drawn symbols. For that, the user draws a stroke, starting inside one symbol and stopping inside another. The stroke is then automatically converted into a link between the symbols. Annotations are textboxes that are ignored for the metamodel creation, since they contain text related to a concrete model.

C. Inferred and User-Defined Symbol Types

Symbols on the drawing canvas can be selected. Upon selection, a context menu includes the option to assign a type (via text input) to the symbol. Each type appears in the type library together with its graphical representation, which is displayed on the right edge of the screen. From there, new instances of types can be created by dragging and dropping them on the drawing canvas. This mechanism is an advantage of having a single environment for both modeling and metamodeling. It gives immediate feedback about all currently defined elements of the language. As the type library allows to re-use defined types, users can get motivated to assign types to elements, even if they do not intend to perform metamodeling [31].

Once some symbols are defined, the type of similar symbols can be inferred. A sketch recognition algorithm recognizes similar, yet untyped symbols. As recognition errors are inevitable, the tool does not automatically assign symbol types. Instead, it displays suggestions to the user in a small popup window. The user can either tap on suggestions or simply ignore them, as they disappear after a couple of seconds. As long as a symbol remains untyped, FlexiSketch internally uses a unique identifier for the symbol type. This is not shown to the user, but needed to distinguish untyped symbols from each other when connection cardinalities are inferred.

D. Inferred and User-Defined Link Types

Each link (edge) in the model represents a connection. Selecting links and assigning types to them works in the same way as for symbols. However, for the appearance of a link, the user has to choose from a predefined set of options (arrow, no arrow, solid line, dashed line, etc.). This allows the tool to guarantee a 1:1 correspondence between semantic constructs and graphical representations of links. If two links have the same appearance, Flexisketch infers that they have the same type, thus prohibiting *symbol overload* [20] for link types. As soon as the user assigns a type to a link, all links with the same appearance in the model automatically get the same type assigned. Conversely, FlexiSketch does not allow the user to assign the same type to two links having different appearances, thus preventing *symbol redundancy* [20] for link types. The restriction of having a 1:1 mapping between link appearances and types also facilitates the inferring of connection cardinalities.

E. Inference of Connection Cardinalities

The user can define cardinalities for a connection type directly by selecting a link of that type on the drawing canvas and using the context menu to set the cardinalities. For connection types having no user-defined cardinalities, FlexiSketch automatically infers them, using a *closed world assumption*: the inference is based only on those links that have been modeled so far. This means that the tool infers very restrictive cardinalities in the beginning, starting with *1..1* when two symbols are connected with a link. When more links of the same type connect the same symbol with others, the cardinality rule is relaxed to, e.g., *1..4*. Thus, the tool never sets a cardinality to *n*; such generalizations must be done by the human user. Alternatively, the inference algorithm could be changed such that *n* is inferred whenever a cardinality is greater than *1*.

The tool infers cardinalities whenever one of the following events happens: (i) The sketched model is saved (the metamodel is saved as well); (ii) The user wants to set the cardinalities of a connection type. Instead of presenting empty fields to the user, they are pre-filled with the inferred cardinalities (unless user-defined cardinalities already exist); (iii) The user locks the metamodel (see Sect. III.G).

As described in Sect. III.A, a connection type is defined as $R(A, B)$, where R is the type of the link, and A, B are the types of the connected symbols. To infer the cardinalities of a connection type, FlexiSketch looks at all occurrences of $R(A, B)$ in the sketch. If a symbol of type A has less or more outgoing links to symbols of type B than defined by the current minimum and maximum outgoing cardinalities, the cardinalities are automatically adjusted accordingly. The same is done for defining the incoming cardinalities.

Cardinalities and connection types can also change when symbol types and/or link types are changed. A typical case is when a single link type is used to connect many untyped symbols. For each link, a connection type needs to be created, as each connected symbol potentially has a different type and different cardinalities. When the user then assigns the same type to several symbols, the according connection types are consolidated into one, and the cardinalities are updated. Rules can also get more restrictive when links are deleted. But cardinalities are never set more restrictively than the values already defined by the user. Deleting a link or a symbol from the drawing canvas can also alter the list of connection types. If a connection type has no more instances on the canvas, it gets deleted. However, link types and symbol types that are defined by the user and visible in the type library are never deleted automatically.

Fig. 2 shows an example. The tool manages two connection types *performs*(*person*, *activity*) and *performs*(*person*, *unknown_type_1*). The type *unknown_type_1* indicates that the user has not assigned a type to this symbol. For *performs*(*person*, *activity*), the tool detects that one symbol of type *person* is connected to at most one symbol of type *activity*, while the other *person* has no connection. It therefore infers the outgoing cardinalities *0..1*. The incoming cardinalities are *1..1*, as each symbol of type *activity* has exactly one incoming *performs* link from a *person*. The cardinalities for the other connection type are identical. If the user now assigns the type

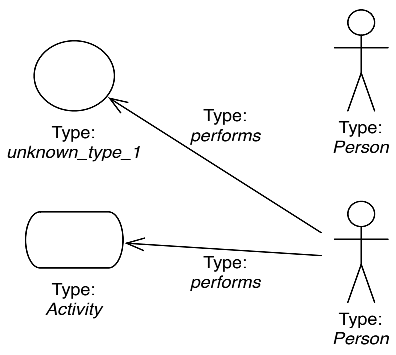


Fig. 2. Inferring example.

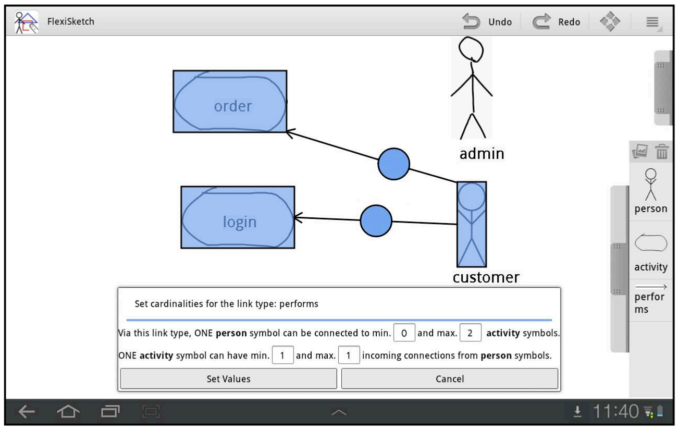
activity to the untyped symbol, the two connection types are merged, because both now define the same connection: *performs*(*person*, *activity*). Since a *person* is now connected to multiple *activities,* the *0..1* cardinality rule is relaxed to *0..2*. If the user deletes one of the *activity* symbols, FlexiSketch checks the rest of the sketch to see whether it has to set the *0..2* cardinalities back to *0..1*. This depends whether there is still another *person* symbol in the sketch that is connected to more than one *activity* symbol or not.

F. The Wizard – Interactive Guidance

In addition to adding types and setting cardinalities by using the context menu icons of sketched elements, FlexiSketch provides a wizard that helps modelers to supply missing metamodeling information. The wizard can be consulted on demand. It is passive in order not to distract the user from the modeling task. The wizard can be especially useful when it is called before saving the finished model sketch to ensure that no metamodel information is missing. Currently, the wizard consists of three steps: first it asks about types of unknown symbols, then about links, and finally about cardinalities for connection types. In each step, the wizard displays a separate page and question per element.

In the first and the second step, the wizard identifies untyped symbols and links respectively. When it detects one, it shows it to the user on the drawing canvas. If needed, the tool scrolls the canvas to make the element visible onscreen and then highlights it with a blue background. At the bottom of the screen, the wizard asks the user to define the type. A definition can be skipped if the user does not regard the currently shown element to be relevant. As soon as the user assigns a type to a particular link, this type is automatically assigned to identical looking links.

In the third step, the wizard looks for connection types where at least one of the four cardinalities is not marked as *user-defined* (the state of a cardinality can be *inferred* or *user-defined*). When it finds one, it randomly picks an instance of it (a link) on the drawing canvas. It highlights the link and the connected symbols and provides the options to *set cardinalities.* Figures 3 and 4 show a screenshot of the wizard asking about cardinalities for a connection type from the inferring example in Sect. III.E. The cardinality values in the fields (*0..2* and *1..1*) were inferred by FlexiSketch and are now presented to the user.



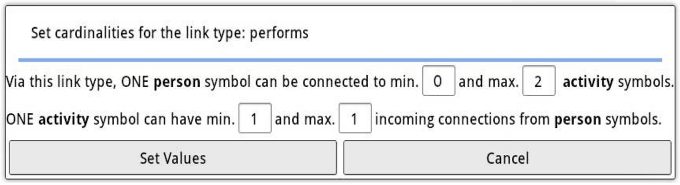
Set cardinalities for the link type: performs

Via this link type, ONE **person** symbol can be connected to min. 0 and max. 2 **activity** symbols.

ONE **activity** symbol can have min. 1 and max. 1 incoming connections from **person** symbols.

Set Values Cancel

Fig. 3. The wizard highlights an instance of a connection type and asks for the cardinalities.



Set cardinalities for the link type: performs

Via this link type, ONE **person** symbol can be connected to min. 0 and max. 2 **activity** symbols.

ONE **activity** symbol can have min. 1 and max. 1 incoming connections from **person** symbols.

Set Values Cancel

Fig. 4. A close-up of the wizard window.

G. Storing Metamodels and the Lock Mechanism

The co-evolution of models and metamodels imposes challenges when it comes to storing the metamodels and how the versioning of metamodels should be handled [29]. In our case, metamodels are undergoing an almost continuous evolution: as a user changes the model, this in turn might also change the corresponding metamodel. Earlier models that had the same underlying metamodel might no longer be compatible with the new metamodel version. We present two mechanisms to minimize the synchronization problem between multiple model sketches and the metamodel.

First, a metamodel is stored together with each sketched model. This ensures that each model conforms to at least one metamodel at any time. If two or more metamodels need to be merged into one, we can create a common metamodel automatically as long as the result is a generalized metamodel, i.e., the merging can be achieved by only adding new meta-information and relaxing existing rules (e.g., changing cardinalities from *1..n* to *0..n*). Such changes to a metamodel belong to the category of so-called *non-breaking changes* [8]. Models conforming to one of the merged metamodels will also conform to the generalized metamodel. There are two more categories: *breaking changes which are resolvable*, and *breaking changes which are not resolvable*. Several researchers [8][9][25] present approaches for handling such metamodel changes.

Second, we introduce a lock mechanism. Metamodels can be saved independently from models. Once a metamodel is thought to be final, it can be locked. Other users can load a locked metamodel and start to sketch a model, but the lock disallows any changes to the metamodel. Therefore it will not be updated according to the model sketch. Instead, parts of the model that do not conform to the metamodel will be highlight-

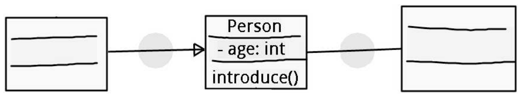


Fig. 5. A minimalistic class diagram fragment.

```
...
 <Symbol>
   <type>Class</type>
   <attributes>
     <labels> ... </labes>
   </attributes>
 </Symbol>
 <Link>
   <type>Association</type>
   <appearance> ... </appearance>
   <direction>bidirectional</direction>
   <connections>
     <connection__1>
       <from__element>Class</from__element>
       <to__element>Class</to__element>
       <from cardinalities>
         <min>0</min> <max>-1</max>
       </from cardinalities>
       <to cardinalities>
         <min>0</min> <max>-1</max>
       </to cardinalities>
     </connection__1>
   </connections>
 </Link>
 <Link>
   <type>Inheritance</type>
   <appearance> ... </appearance>
   <direction>unidirectional</direction>
   <connections>
     <connection__1>
       <from__element>Class</from__element>
       <to__element>Class</to__element>
       <from cardinalities>
         <min>0</min> <max>-1</max>
       </from cardinalities>
       <to cardinalities>
         <min>0</min> <max>1</max>
       </to cardinalities>
     </connection__1>
   </connections>
 </Link>
...
```

Fig. 6. Metamodel of the fully defined class diagram fragment from Fig. 5.

ed accordingly. This mechanism allows companies to leave metamodels unlocked as long as they are building DSLs. They can lock a metamodel to finalize their DSL, signaling modelers that they now have to adhere to the metamodel (unless companies want to unlock it again for improving the DSL).

Finally, metamodels can be saved and exported at any point in time. We illustrate this functionality with a small, sketched class diagram fragment as shown in Fig. 5. The user has drawn three boxes and assigned the type *class* to them. She connected the box in the middle and the one on the right side with a link and defined it as *association*. Then she connected the third box with a link, changed its appearance to a line with an arrow head, and defined it as *inheritance*. The user then defined the cardinalities for the two connection types, and added some text to one of the boxes. Figure 6 shows an excerpt of the corresponding metamodel generated by FlexiSketch.

IV. Initial Evaluation Results

We investigated to what extent FlexiSketch supports novice modelers in providing metamodel information for their model sketches. We conducted an experiment with 31 second term computer science students. The students had no prior metamodeling knowledge, and only little experience in modeling. After a tutorial in which the students learned about the tool functionalities, the students were assigned a use case modeling task. No introduction to metamodeling was given. However, the handouts stated that all model elements should be defined because the tool must be able to interpret the sketched diagrams. An online questionnaire completed the experiment.

Results show that the students were able to correctly define symbol and relationship types. In contrast, many students made mistakes in the cardinality definitions. The reason was that these students were thinking on the model level instead of the metamodel level, thus trying to assign cardinalities to individual relations instead of relationship types.

We performed this experiment with students to prove that FlexiSketch is easy to use and even modeling novices can generate metamodels with it. However, results suggest that users need at least some basic metamodeling knowledge in order to master metamodeling tasks that go beyond type assignment. Future studies will focus on requirements engineers, who are also the main target group of FlexiSketch.

V. Related Work

We identified related work about sketching and design in software engineering as well as metamodel inference. However, we are not aware of any work within the SE field that tries to combine a lightweight modeling approach (in this case model sketching) with a user-friendly metamodeling solution in a single tool. Most work about metamodeling focuses on the technical aspects, assuming that a metamodeling expert learns how to operate a formal modeling tool. The aspects of usability and user-friendliness are ignored.

The idea of bringing sketch interfaces and recognition into play to foster creativity and facilitate design tasks is not new [11]. [3] presents a generic approach to generating diagram editors which support and analyze hand drawings. Several other researchers have incorporated a sketch interface into their semi-formal modeling tools, e.g., MaramaSketch [14], InkKit [23], and SketchREAD [1]. [16] gives a broad overview of similar approaches. While some of these approaches might allow users to alter the concrete notation, they only support predefined modeling languages. In contrast, the Calico tool [19] focuses on supporting an informal form of software design that heavily relies on sketching. It provides some means of structuring the sketches, but does not allow to formalize them. [10] and [28] discuss a step-wise formalization of models. New modeling languages can be created by linking artifacts of already existing languages.

Several research tackles metamodel creation from model examples, e.g., [13]. MARS [15] is a tool for reconstructing missing metamodels for a given set of models. Cuadrado et al. [9] propose an interactive, bottom-up metamodeling approach similar to ours. But modeling and metamodeling cannot be performed in the same tool. Cho et al. [7] focus on technical aspects of incremental and iterative metamodel definition by providing model examples. User interaction and tool-support are not discussed. Design guidelines for DSLs can be found in [17] and [22]. [6] and [26] discuss design patterns for metamodels. Regarding user guidance, Qattous et al. [24] demonstrate that defining metamodel constraints with a by-example approach outperforms a form-based wizard approach.

VI. Conclusions and Future Work

We have shown how FlexiSketch supports semi-automatic step-wise creation of metamodels, without the help of metamodeling experts. Following our research objectives given in Sect. II.A, we presented strategies on how to gather information relevant for metamodel generation, using both automated model inferring and wizard-based user guidance. A key contribution of our work is our technical solution, which has been implemented in the FlexiSketch prototype. Experiment results highlight that modelers are able to provide relevant metamodel information when working with FlexiSketch.

FlexiSketch provides a lightweight and user-friendly approach to modeling and metamodeling. As we are mainly concerned about gathering basic metamodel information, we do not focus on building high-quality, sophisticated metamodels. Readers who are interested in the latter are referred to [6][22][17][26].

In its current version, FlexiSketch does not support attributes (just textboxes as child elements of other elements), the nesting of symbols (containment), abstract classes, and inheritance in the metamodel. It also does not support any spatial information, e.g., there is no difference whether a symbol is placed to the right or to the left of another symbol. Feedback from practitioners about these missing properties suggests that containment and attributes are strongly needed, whereas inheritance and spatial information are less important features.

We are working on extending FlexiSketch with the missing features, especially attributes and containment, and plan to improve the wizard for better user-guidance. An export function will allow exporting the generated metamodels to MetaEdit+ [27]. We will also focus on the continuous use of FlexiSketch, which requires novel features regarding metamodel versioning. Other future work focuses on real-world case studies where practitioners use FlexiSketch within their daily work.

References

[1] C. Alvarado and R. Davis, “SketchREAD: a multi-domain sketch recognition engine”, in 17th ACM Symp. User Interface Softw. Techn. (UIST 2004), pp. 23–32, 2004.

[2] C. Atkinson and T. Kühne, “Model-driven development: a metamodeling foundation”, in IEEE Software 20(5):36–41, 2003.

[3] F. Brieler and M. Minas, “Recognition and processing of hand-drawn diagrams using syntactic and semantic analysis”, in Work. Conf. Adv. Visual interfaces (AVI 2008), pp. 181–188, 2008.

[4] A. W. Brown, “Model driven architecture: principles and practice”, in Software and Systems Modeling 3(4):314–327, 2004.

[5] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, “Let’s go to the whiteboard: how and why software developers use drawings”, in CHI 2007 Conf., pp. 557–566, 2007.

[6] H. Cho and J. Gray, “Design Patterns for Metamodels”, in Proc. Compilation of the co-located workshops on DSM, TMC, AGERE!, AOOPES, NEAT, VMIL, pp. 25–32, 2011.

[7] H. Cho, J. Gray, and E. Syriani, “Creating Visual Domain-Specific Modeling Languages from End-User Demonstration”, in 4th Int’l. Workshop Modelling in Softw. Eng. at ICSE 2012, pp. 22–28, 2012.

[8] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, “Automating co-evolution in model-driven engineering”, in 12th Enterprise Distr. Obj. Comp. Conf. (EDOC 2008), pp. 222–231, 2008.

[9] J. S. Cuadrado, J. de Lara, and E. Guerra, “Bottom-up meta-modelling: an interactive approach”, in 15th MODELS Conf., pp. 3–19, 2012.

[10] J. R. Douglass, “Language of Languages for Flexible Development”, in SPLASH Workshop on Flexible Modeling Tools, 2010. http://www.ics.uci.edu/~nlopezgi/flexitools/papers/douglass_flexitools_splash2010.pdf [last checkd 05/17/13]

[11] T. O. Ellis, J. F. Heafner, and W. L. Sibley, “The grail project: an experiment in man-machine communications”, RAND Memorandum RM-5999-ARPA, 1969. http://www.rand.org/content/dam/rand/pubs/research_memoranda/2005/RM5999.pdf [last checked 05/17/13]

[12] R. Ferguson and A. Hunter, “MetaBuilder: the diagrammer's diagrammer”, in 1st Int’l. Conf. Theory and Application of Diagrams, pp. 407–421, 2000.

[13] G. Gabrysiak, H. Giese, A. Lüders, and A. Seibel, “How Can Metamodels Be Used Flexibly?” in ICSE Workshop on Flexible Modeling Tools, 2011. http://www.ics.uci.edu/~nlopezgi/flexitoolsICSE2011/papers/gabrysiak_flexitools_icse2011.pdf [last checkd 05/17/13]

[14] J. Grundy and J. Hosking, “Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool”, in 29th Int’l. Conf. Softw. Eng., (ICSE 2007), pp. 282–291, 2007.

[15] F. Javed, M. Mernik, J. Gray, and B. R. Bryant, “MARS: a metamodel recovery system using grammar inference”, Inf. Softw. Technol. 50(9–10):948–968, 2008.

[16] G. Johnson, M. D. Gross, J. Hong, and E. Yi-Luen Do, “Computational support for sketching in design: a review”, Foundations and Trends in Human–Computer Interaction 2(1):1–93, 2009.

[17] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schindler, and S. Völkel, “Design Guidelines for Domain Specific Languages”, in OOPSLA workshop DSM, pp. 7–13, 2009.

[18] A. Kleppe, Software language engineering: creating domain-specific languages using metamodels. Addison-Wesley, 2008.

[19] N. Mangano, A. Baker, M. Dempsey, E. Navarro, and A. van der Hoek, “Software design sketching with Calico”, in 24th Int’l Conf. Autom. Softw. Eng. (ASE 2010), pp. 23–32, 2010.

[20] D. L. Moody, “The physics of notations: toward a scientific basis for constructing visual notations in software engineering”, in IEEE TSE 35(6):756–779, 2009.

[21] H. Ossher, A. van der Hoek, M.-A. Storey, J. Grundy, and R. Bellamy, “Workshop on flexible modeling tools (FlexiTools2010)”, in 32nd Int’l. Conf. Softw. Eng., (ICSE 2010), pp. 441–442, 2010.

[22] R. F. Paige, J. S. Ostroff, and P. J. Brooke, “Principles for modeling language design”, Inf. Softw. Technol., 42(10):665–675, 2000.

[23] B. Plimmer and M. Apperley, “INTERACTING with sketched interface designs: an evaluation study”, in CHI 2004 Conf., pp. 1337–1340, 2004.

[24] H. Qattous, P. Gray, and R. Welland, “An empirical study of specification by example in a software engineering tool”, in 4th Int’l. Symp. Empirical Softw. Eng. Measur. (ESEM 2010), 2010.

[25] L. M. Rose, D. S. Kolovos, R. F. Paige, and F. A. C. Polack, “Enhanced Automation for Managing Model and Metamodel Inconsistency”, in 23th Int’l Conf. Autom. Softw. Eng. (ASE 2009), pp. 545–549, 2009.

[26] C. Schäfer, T. Kuhn, and M. Trapp, “A Pattern-based Approach to DSL Development”, in Proc. Compilation of the co-located workshops on DSM, TMC, AGERE!, AOOPES, NEAT, VMIL, pp. 39–46, 2011.

[27] J.-P. Tolvanen and S. Kelly, “MetaEdit+: defining and using integrated domain-specific modeling languages”, in OOPSLA, pp. 819–820, 2009.

[28] B. Volz, M. Zeising, and S. Jablonski, “The open meta modeling environment”, in ICSE Workshop on Flexible Modeling Tools, 2011. http://www.ics.uci.edu/~nlopezgi/flexitoolsICSE2011/papers/volz_flexitools_icse2011.pdf [last checked 05/17/13]

[29] G. Wachsmuth, “Metamodel Adaptation and Model Co-adaptation”, in 21st Eur. Conf. Obj.-Orient. Progr. (ECOOP 2007), pp. 600–624, 2007.

[30] D. Wüest, N. Seyff, and M. Glinz, “Flexible, lightweight requirements modeling with FlexiSketch”, in 20th Int’l. Req. Eng. Conf (RE’12), pp. 323–324, 2012.

[31] D. Wüest, N. Seyff, and M. Glinz, “FlexiSketch: a mobile sketching tool for software modeling”, in 4th MobiCASE Conf. , pp. 225–244, 2013.