2014 Second IEEE Working Conference on Software Visualization

Slicing-based Techniques for Visualizing Large Metamodels

Arnaud Blouin
INSA Rennes
Rennes, France
arnaud.blouin@irisa.fr

Naouel Moha
University of Québec at Montréal
Montréal, Canada
moha.naouel@uqam.ca

Benoit Baudry
Inria
Rennes, France
benoit.baudry@inria.fr

Houari Sahraoui
University of Montréal
Montréal, Canada
sahraoui@iro.umontreal.ca

***Abstract*—In model-driven engineering, a model describes an aspect of a system. A model conforms to a metamodel that defines the concepts and relationships of a given domain. Metamodels are thus corner-stones of various meta-modeling activities that require a good understanding of the metamodels or parts of them. Current metamodel editing tools are based on standard visualization and navigation features, such as physical zooms. However, as soon as metamodels become larger, navigating through large metamodels becomes a tedious task that hinders their understanding. In this work, we promote the use of model slicing techniques to build visualization techniques dedicated to metamodels. We propose an approach based on model slicing, inspired from program slicing, to build interactive visualization techniques dedicated to metamodels. These techniques permit users to focus on metamodel elements of interest, which aims at improving the understandability. This approach is implemented in a metamodel visualizer, called `Explen`.**

I. Introduction

The fundamental idea of Model-Driven Engineering (MDE) is to consider models as first-class entities. A model conforms to a metamodel that describes the concepts of a given domain. Metamodels, represented graphically as class diagrams, are thus corner-stones of various meta-modeling activities that require a good understanding of the metamodels or parts of them. Such activities consist of, for instance, transforming models into code, creating editing tools for a metamodel, or comparing metamodels. The current mainstream metamodel editors, such as `EcoreTools` provided by the Eclipse Modeling Framework (EMF)[1], however, offer basic interactive features to navigate through metamodels (physical zoom, scroll bars, *etc.*). As soon as metamodels become larger, understanding and manipulating metamodels becomes a tedious task using these basic interactive features. For instance, Figure 1 is an overview of the metamodel UML [1] obtained using the physical zoom of `EcoreTools`. Many classes are gathered and reduced so that identifying one class or its relations with other ones becomes awkward.

When modelers are interested only in a specific part of a metamodel, they may want to focus on such a specific part by, for instance, hiding the rest of the metamodel. For instance, for the visualization of a metamodel, a modeler may be interested in semantic relationships between classes such as the inheritance tree of a given class or the classes linked by a composition reference to a given class. With the current editors, modelers are forced to manually and astutely combine sequences of filtering and navigation primitive operations to rebuild these parts of interest. This manual exploration task may be time-consuming and error-prone.

Visualization techniques are broadly used in software engineering and have proven their usefulness for software comprehension and in particular, interactive visualization that provides meaningful navigation capabilities [2]. Gračanin *et al.* summarized the benefits in terms of comprehension brought by software visualization to different domains [3]. Closely, previous work on UML class diagrams highlights the research interest on improving the understanding of class diagrams [4], [5], [6]. Our contribution is twofold: we show that model slicing can be used to build interactive visualization techniques dedicated to metamodels; we propose a software engineering approach to ease the development of model slicers to be used within metamodel visualizers as a visualization engine. Model slicing permits users to focus on metamodel elements of interest, which aims at improving the understandability. We developed a metamodel visualizer, namely `Explen`, embedding slicing-based interactive visualization techniques.

The paper is organized as follows. Section II introduces model-driven engineering. Section III depicts a motivating scenario. Section V describes how model slicing can be used to develop interactive visualization techniques for improving the understandability of large metamodels. The paper ends with the related work (Section IV) and the future work (Section VI).

II. Model-Driven Engineering

The traditional way scientists use to master complexity is to resort to modeling. According to Jeff Rothenberg, "*Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality*" [7]. In engineering, one wants to break down a complex system into as many models as

[1] http://www.eclipse.org/modeling/emf/

978-0-7695-5305-4/14 $31.00 © 2014 IEEE
DOI 10.1109/VISSOFT.2014.14

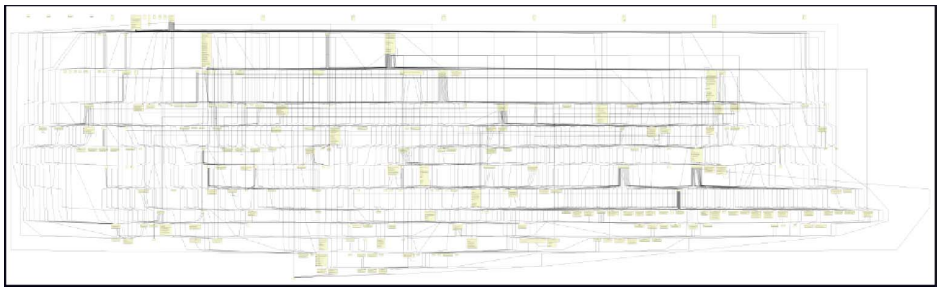

Fig. 1. Bird view of the UML metamodel using `EcoreTools` (246 classes and 769 relationships)

needed to address all the relevant concerns in such a way that they become understandable enough. The fundamental idea of MDE [8] is to consider models as first-class entities in software-development processes. MDE aims at reducing the complexity associated with developing complex software systems. In MDE, a model describes an aspect of a system and is typically created or derived for specific development purposes. Each model conforms to a well-defined metamodel that describes the concepts and relationships of a given domain. For instance, Figure 2 depicts an excerpt of the Ecore metamodel[1]. This metamodel defines the concepts of classes (*EClass*) having references (*EReference*), attributes (*EAttribute*), and operations (*EOperation*). Metamodels are corner-stones of various meta-modeling activities that require a good understanding of the metamodels or parts of them. As illustrated in Figure 2, the meta-modeling conventions promote a 2D class diagram representation, close to the UML class diagram, to represent metamodels graphically. Models, conforming to their metamodel, can be manipulated by model transformations for various purposes (*e.g.* code generation).

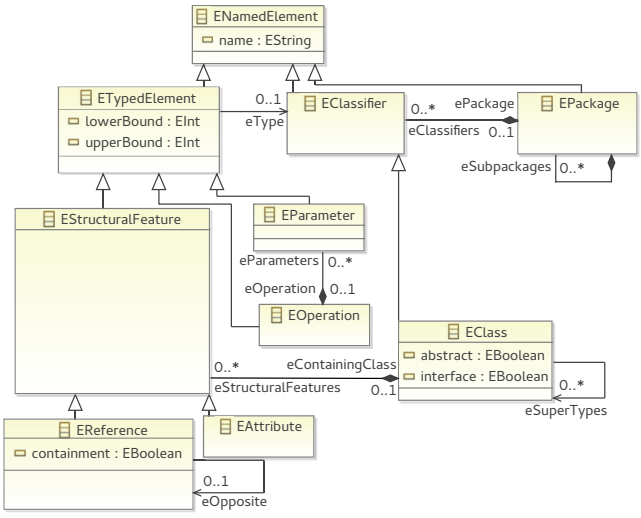

Fig. 2. An example of metamodel: an excerpt of the Ecore metamodel

III. Motivating Scenario

We motivate the need for integrating slicing-based interactive visualization features within graphical modeling tools based on the following common scenario. In an illustrative purpose, this scenario is based on the UML metamodel. Our approach, however, is dedicated to any large metamodel.

Scenario. A modeler has to write a model transformation that generates Java code from UML models. The modeler has already a rough idea of main classes of the UML metamodel required for the transformation. These classes include: *Association*, *Class*, *Package*, *Parameter*, *Property*, and *Operation*. However, before writing the transformation, the modeler needs to have a clear and precise understanding of how these classes are organized within the UML metamodel and identify the properties and operations required for the transformation. To acquire this understanding, the modeler visualizes the UML metamodel using, for example, `EcoreTools` of EMF. The visualization of the whole UML metamodel is not a great help to her: as illustrated in Figure 1, the UML metamodel is overcrowded because of its 246 classes and 769 relationships. Yet, the modeler navigates and explores the UML metamodel to identify precisely classes and elements (properties and operations) required for her transformation. The following tasks are examples of such a process of navigation and exploration:

Task 1. The modeler uses the physical zoom of the editor to focus on classes related to *Class*. Since all the classes directly linked to *Class* do not appear, and a high number of relationships are tangled, the interest of this view is limited for the modeler. Zooming out again will gather too many classes and the view will still be unreadable such as in Figure 1.

Task 2. The modeler explores the inheritance relationships of *Class* to see all elements this class inherits. However, this task is quite difficult because of the tangled relationships and the high number of classes that hinder the visibility. Moreover, the multi-inheritance of several UML classes complicates the navigation within the inheritance tree of *Class*. For example, the inheritance tree of *Class* is composed of 17 inheritance relations. To get all the inherited properties of *Class*, the modeler needs to navigate through each of these classes.

Task 3. The modeler wants to obtain a reduced view of the UML metamodel that contains only classes relevant for her transformation. The modeler hides irrelevant metamodel

elements using the filtering and navigation capabilities of EcoreTools. For instance, the filter *Hide Selection* enables the modeler to hide all the selected elements. After having hidden one by one the 216 classes of the metamodel not concerned by the transformation, the modeler finally obtains a subset of the UML metamodel that only contains the 30 relevant classes for her transformation.

IV. Related Work

Metamodels are represented graphically using 2D class diagrams, close to the UML class diagram notation. So, research works on visualizing UML class diagrams or classes are related and could help to perform the previous scenario. Various works have been conducted on layouts to propose new algorithms for minimizing crossing relations [9], [10]. Different guidelines for drawing class diagrams have also been proposed [11]. Several research works proposed to represent class models differently than using class diagrams, such as the class blueprint [12]. Our work, however, focuses on supplementing standard metamodel editors with interactive visualization techniques. Also, we keep the focus on the class diagram representation promoted and widely-used within the MDE community. Complementary to our work, techniques have been proposed to visualize large UML class diagrams based on focus+context techniques [13], [14]. One difference compare to these works is our use of a language for defining model slicers. It aims at easing the development of several visualization features to be integrated into metamodel editors.

Reducing the amount of visible data to a subset of interest using filters and zooms have been already proposed for graph visualization [15], [16], [17]. In our work, we consider the structural concepts defining metamodels (composition, inheritance, *etc.*) to build the interactive visualization techniques.

Regarding MDE activities, research works have been proposed to ease the development of model visualizers and their layout [18], or to use gestures within modeling editors [19]. Our work follows this trend that aims at easing the development process of modeling editors and completing them with advanced interactive navigation features.

V. Slicing-based Techniques for Interactively Visualizing Large Metamodels

A. On the benefits of model slicing to build filtering features

Ideally, the modeler, involved in the previous scenario, would have preferred to obtain the views in a more straightforward way instead of astutely combining the editor's filtering and navigation capabilities. If the editor had provided a filtering capability to show only classes directly linked to a selected class, Task 1 might have been easier. A similar filtering capability for the inheritance relationships might have eased Task 2. Regarding Task 3, a more complex filtering capability that combines the two previous ones might have also been useful for the modeler.

Model slicing is a model comprehension technique inspired by program slicing. The model slicing process involves extracting a subset of model elements that represent a *model slice*. The model slice may vary depending on the intended purpose. For example, when seeking to understand a large metamodel, it may help to extract the sub-part of the metamodel that includes only the dependencies of a particular class.

In our previous work [20], [21], we proposed Kompren, a domain-specific language to define model slicers. A *model slicer* expressed with Kompren refers to a set of classes and relations from the *metamodel* under study. Instances of the referenced classes and relations will be sliced (*i.e.* extracted) from the *input model*. Listing 1 gives an example of a Kompren slicer. This slicer aims at slicing models of the Ecore metamodel (depicted in Figure 2). It selects several classes, references, and attributes of this metamodel to be sliced. This slicer is fully described in the next sub-section.

```
1  slicer MetamodelSlicer {
2    domain: "org.eclipse.emf.ecore/model/Ecore.genmodel"
3    input: EClass
4    radius: EClass
5    slicedClass: ENamedElement
6    slicedClass: EStructuralFeature feat
7      constraint: card1 [[ feat.lowerBound>0 ]]
8    slicedProperty: EClass.eSuperTypes option
9    slicedProperty: EClass.eSuperTypes option opposite(subTyp)
10   slicedProperty: EClass.eStructuralFeatures option
11   slicedProperty: EClass.eOperations option
12   slicedProperty: ETypedElement.eType
13 }
```

Listing 1. The Kompren model slicer used for visualizing metamodels

After being manually specified by developers, a Kompren model slicer is compiled as Java code to be integrated in a Java program as a library. The execution of such an executable slicer program takes as input a model (here an Ecore model), instance of the input metamodel, and slicing criteria. Slicing criteria are model elements from the input model (here elements of an Ecore model) that provide the entry point for extracting a model slice. The slicing process visits all the elements specified in the model slicer starting from the slicing criteria. When visited, an element is *sliced* (*i.e.* extracted). In our case of metamodel visualization, it means that the sliced elements are displayed while the rest is hidden. This process permits to define dynamic filtering features to explore large metamodels, as detailed in the next sub-section.

B. Explen: a Kompren-Based Metamodel Visualization Tool

We developed a 2D metamodel visualizer in Java, called Explen, embedding dedicated interactive visualization techniques we developed using Kompren: we defined the model slicer described in Listing 1. Then, it has been integrated into Explen as the visualization engine. So, when users use Explen's features, described in this section, this slicer is called to perform a slicing and notify Explen about the metamodel elements to hide or show.

Semantic zoom. The physical zoom is supplemented with a semantic zoom that shows different metamodel elements depending on the zoom level. When zooming out at 50% the attributes, operations, and roles are no more displayed. The goal of this feature is to lighten or complete the amount of displayed information when visualizing at a given zoom level. This semantic zoom also works thanks to the model slicer

depicted in Listing 1. `Explen` also provides the following dynamic filters to be applied on selected classes.

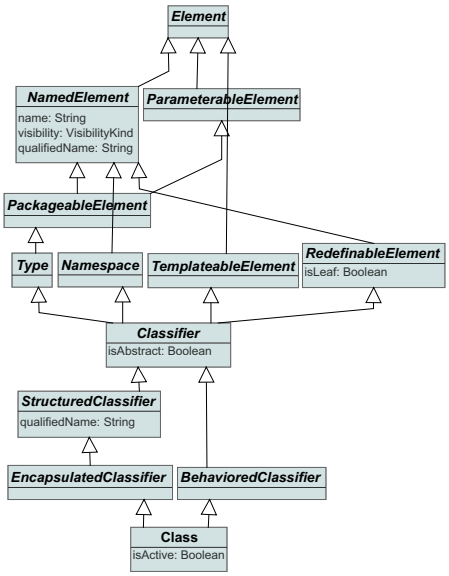

Fig. 3. The super inheritance tree of the UML class `Class`

The super and lower inheritance filters. They show the super or lower inheritance tree of the targeted class. These two filters can be parameterized with one option, the radius effect. When activated with a given value greater than 0, this option shows the classes in relation with the targeted class by a distance lower than or equals to the radius value. For instance, when the radius equals 1, only the direct classes of the targeted class are displayed. When set to 2, only these direct classes and their respective direct classes are shown. This option permits to reduce the number of classes shown in the view. For instance, the goal of the task 2 is to get *all* elements of a class (*i.e.*, inherited and intrinsic properties, and operations). The modeler performs this task by getting the hierarchical tree of *Class* as depicted in Figure 3. Due to constraints on space, the classes' operations have been hidden, and the classes have been re-layouted manually to reduce the spacings.

The slicing filter. It hides classes not in relation (inheritance or references) with the targeted class. This filter has three options that can be combined. The first option slices composition references only. The second option slices references having their minimal cardinality greater than 0 only. The third option is the radius parameter. For instance, the task 3 consists of showing only classes in relation with the UML class `Class`. This task can be performed using the `Explen`'s slicer. To show only classes closely related to the class `Class`, the radius is set to 3. We also configure the slicer to consider composition references only. Then, the slicer is applied on the class `Class` and the classes not sliced are hidden. Figure 4 shows the result of this slicing where only 18 classes among the 246 others are displayed. The resulting classes have been manually re-layouted.

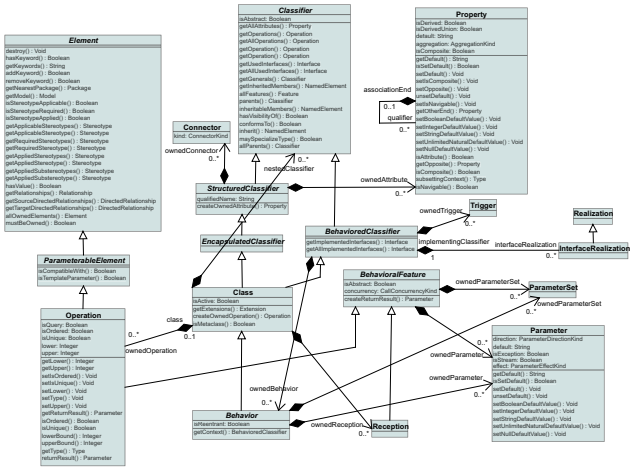

Fig. 4. Slicing the UML metamodel using the class `Class` as input and parameterized with a radius of 3 and by slicing composition references only

The flattening filter. The super hierarchy of the targeted class is removed to move into this last all its inherited attributes and relations. To perform the task 2, the modeler can also flatten this hierarchy to put in *Class* all the properties and operations of its super-classes. Figure 6 shows the result of the flattening of *Class*. All the super-classes of *Class* have been removed while their properties and operations have been moved into *Class*. The filters can also be successively combined. For instance, the goal of the task 1 is to show classes in direct relationship with *Class*. The modeler can accomplish this task with our viewer by restricting the radius effect of the slicer using the user interface: when the radius effect is set to 1, only classes in direct relationship with the sliced class are shown. Figure 6 also illustrates such successive combinations where the flattening filter is followed by a slicing of `Class` parameterized with a radius of 1.

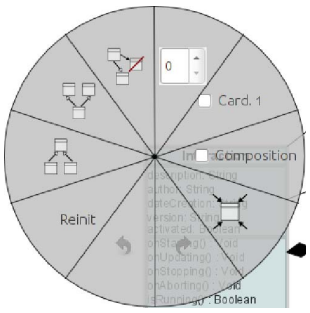

Fig. 5. The piemenu activated by clicking on a class to apply filters.

Users can apply these filters by right-clicking on a class to display a piemenu (Figure 5). The buttons of this menu permit to customize and apply filters on the targeted class used as a slicing criterion.

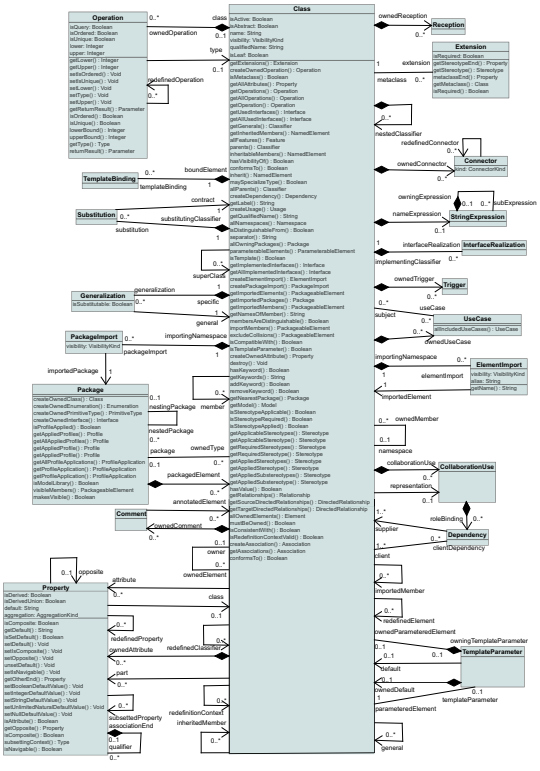

Fig. 6. Flattening of the UML class *Class* followed by a slicing parameterized with a radius of 1

These filters (super and lower inheritance, slicing, and flattening) have been developed using `Kompren` with the slicer given in Listing 1 and integrated into our Java application `Explen` as a visualization library. Since the visualized metamodels are defined using the Ecore meta-metamodel (Figure 2), the developed model slicer is based on this last (line 2). These filters work when applied on a selected class. The slicing criteria of the model slicer are classes (`EClass`, line 3). All the classes of type `ENamedElement` (line 5) are sliced. The references that the slicer goes through during the slicing process to extract instances of `ENamedElement` are defined from the line 8 to 12. The reference `eSuperTypes` is used to extract both the super and lower inheritance tree of a given class (lines 8-9). The keyword `option` identifies the corresponding sliced element as optional. This feature is used to define in a single model slicer, multiple filtering features. The reference `eStructuralFeatures` (line 10) defines the references and the attributes of a class, while `eOperations` (line 11) refers to their operations. The reference `eType` (line 12) is considered in our case as the targeted class of a reference. The sliced class `EStructuralFeature` is supplemented with a constraint (line 7). Constraints are optional conditions that must be respected to trigger the slicing of the element targeted by the condition. In our case, this constraint permits to slice references having a minimal cardinality greater than 0. It is used by the slicing filter that users can parameterize through the user interface. Finally, a radius is defined on the class `EClass` (line 4) as a parameter of lower/super inheritance and slicing filters.

VI. Conclusion and Future Work

In this work, we promote the use of model slicing techniques to build visualization techniques dedicated to metamodels. We implemented a metamodel visualizer, called `Explen`. The `Explen`'s visualization features were developed using `Kompren`, a domain-specific language for defining model slicers. We show how to develop different kinds of filters using `Kompren` to improve the understandability of metamodels.

In our future work, we will conduct an empirical study to assess the benefits of the proposed approach when doing metamodeling tasks involving metamodels. We also think that this approach can be adapted to UML class diagrams.

References

[1] OMG, "UML 2.1.1 Specification," 2007. [Online]. Available: http://www.omg.org/technology/documents/formal/uml.htm

[2] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Journal of Systems and Software*, pp. 171–185, 1999.

[3] D. Gračanin, K. Matković, and M. Eltoweissy, "Software visualization," *Innovations in Systems and Software Engineering*, pp. 221–230, 2005.

[4] Y.-G. Guéhéneuc, "TAUPE: towards understanding program comprehension," in *Proc. of CASCON'06*, 2006.

[5] S. Yusuf, H. Kagdi, and J. Maletic, "Assessing the comprehension of UML class diagrams via eye tracking," in *Proc. of ICPC'07*, 2007, pp. 113–122.

[6] G. Cepeda Porras and Y.-G. Guéhéneuc, "An empirical study on the efficiency of different design pattern representations in UML class diagrams," *Empirical Softw. Eng.*, vol. 15, no. 5, pp. 493–522, 2010.

[7] J. Rothenberg, *The Nature of Modeling*. John Wiley and Sons, 1989.

[8] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[9] J. Seemann, "Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams," in *Graph Drawing*, 1997.

[10] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel, "A new approach for visualizing UML class diagrams," in *Proc. of SoftVis'03*. ACM Press, 2003, p. 179.

[11] H. Eichelberger and K. Schmid, "Guidelines on the aesthetic quality of UML class diagrams," *Information and Software Technology*, vol. 51, no. 12, pp. 1686–1698, 2009.

[12] S. Ducasse and M. Lanza, "The class blueprint: Visually supporting the understanding of classes," *IEEE Transactions on Software Engineering*, vol. 31, pp. 75–90, 2005.

[13] H. Kagdi and J. Maletic, "Onion graphs for focus+context views of UML class diagrams," in *Proc. of VISSOFT'07*, 2007, pp. 80–87.

[14] M. Frisch and R. Dachselt, "Off-screen visualization techniques for class diagrams," in *Proc. of SOFTVIS'10*, 2010, p. 163.

[15] H. P. Kumar, C. Plaisant, and B. Shneiderman, "Browsing hierarchical data with multi-level dynamic queries and pruning," *International Journal of Human-Computer Studies*, vol. 46, no. 1, pp. 103 – 124, 1997.

[16] T. Christian, J. Abello, F. van Ham, and H. Schumann, "Fisheye tree views and lenses for graph visualization," in *Proc. of IV'06*, 2006.

[17] C. Plaisant, J. Grosjean, and B. Bederson, "Spacetree: supporting exploration in large node link tree, design evolution and empirical evaluation," in *Proc. of INFOVIS'2002*, 2002, pp. 57–64.

[18] H. Fuhrmann and R. von Hanxleden, "Taming graphical modeling," in *Proc. of MODELS'10*, 2010.

[19] A. Scharf and T. Amma, "Dynamic injection of sketching features into GEF based diagram editors," in *Proc. of ICSE'13*, 2013, pp. 822–831.

[20] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux, "Kompren: Modeling and generating model slicers," *Software and Systems Modeling (SoSyM)*, pp. 1–17, 2012.

[21] ——, "Modeling model slicers," in *Proc. of MODELS'11*, 2011, pp. 62–76.