# Augmenting graphical modeling workbenches with semantic-aware interactive features

Théo Giraudet, Arnaud Blouin, Benoit Combemale, Mélanie Bats, Pierre-Charles David

▶ **To cite this version:**

Théo Giraudet, Arnaud Blouin, Benoit Combemale, Mélanie Bats, Pierre-Charles David. Augmenting graphical modeling workbenches with semantic-aware interactive features. Proceedings of the ACM on Human-Computer Interaction , In press, pp.29. 10.1145/nnnnnnn.nnnnnnn . hal-04931045

**HAL Id: hal-04931045**
**https://inria.hal.science/hal-04931045v1**

Submitted on 5 Feb 2025

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

# Augmenting graphical modeling workbenches with semantic-aware interactive features

THÉO GIRAUDET, Obeo, Univ Rennes, IRISA, Inria, France
ARNAUD BLOUIN, Univ Rennes, INSA Rennes, IRISA, Inria, France
BENOIT COMBEMALE, Univ Rennes, IRISA, Inria, France
MÉLANIE BATS, Obeo, France, France
PIERRE-CHARLES DAVID, Obeo, France, France

Domain-Specific Modeling Languages (DSMLs) usually come with a dedicated integrated environment called a modeling workbench. In the context of graphical DSMLs, such environments provide modelers with dedicated interactive features that help them perform navigation and editing tasks. Many of these features are generic and can be used by graphical DSMLs without any specialization (*e.g.,* a physical zoom). Others require specializations in accordance with the involved DSML. For instance, a semantic zoom requires specifying which elements of the model must be graphically modified at the different zoom levels. However, current language workbenches do not propose facilities to help language designers in developing such semantic-aware features for their graphical modeling workbenches. So language designers must develop these features by hand, which is a complex and time-consuming task. This paper proposes a novel approach to help language designers in this task. In addition to existing DSML concerns, such as the syntaxes, we propose to capture the interactive features of the targeted modeling workbench in the form of DSML pragmatics. We propose an implementation of our proposal within one industrial language workbench, Sirius Web. We evaluate our proposal through two representative use cases that support discussion of the feasibility of the proposal. We also evaluate its scalability. The evaluation brings forward challenges the community has to consider while developing highly interactive modeling workbenches.

CCS Concepts: • **Human-centered computing → User interface programming**; • **Software and its engineering → Domain specific languages**; **Integrated and visual development environments**.

Additional Key Words and Phrases: language workbench, usability, graphical modeling language, DSML, interactive feature

**ACM Reference Format:**
Théo Giraudet, Arnaud Blouin, Benoit Combemale, Mélanie Bats, and Pierre-Charles David. 2025. Augmenting graphical modeling workbenches with semantic-aware interactive features. *Proc. ACM Hum.-Comput. Interact.* 1, EICS (January 2025), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

A Domain-Specific Modeling Language (DSML) is a modeling language specifically tailored to a given concern [21, 40]. Developing a DSML also implies developing a dedicated modeling workbench, *i.e.,* a modeling environment in which a modeler can edit models of the DSML. Those modeling workbenches play a central role in the usage of DSMLs as they provide modelers with a wide range

Authors' addresses: Théo Giraudet, Obeo, Univ Rennes, IRISA, Inria, France; Arnaud Blouin, Univ Rennes, INSA Rennes, IRISA, Inria, France; Benoit Combemale, Univ Rennes, IRISA, Inria, France; Mélanie Bats, Obeo, France, France; Pierre-Charles David, Obeo, France, France.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2573-0142/2025/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

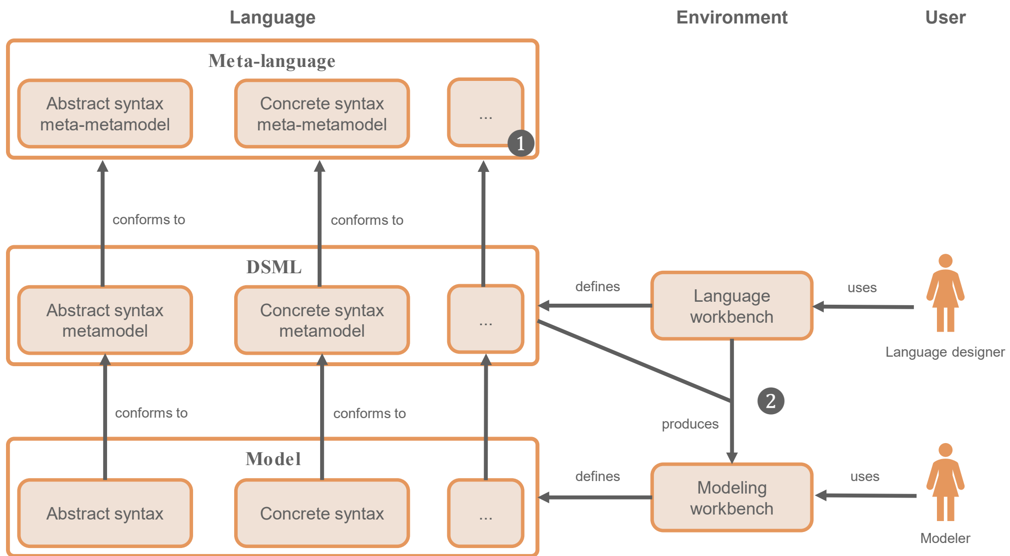
of services and interactive features that improve their usability [38]. Classically, DSML services take the shape of, for example, specialized editors, formatters, consistency checkers, or modeling assistants [49, 51, 53]. Interactive features help users of those modeling workbenches to perform navigation and editing tasks [11, 39, 60]. Several studies have identified the usability of modeling workbenches as a key issue. In [45, 54], the questioned experts are concerned about the usability of the editors they develop or use. In [5, 38, 73], the experts interviewed reported that creating and editing models is slow and that modeling tools are too expensive. Moreover, according to [57], interactive features are important for experts to remember contextual information to identify and fix errors. As a result, researchers are working on user experience and usability in MDE [1], and in particular in language engineering [33, 43, 67]. One identified challenge is the need for customizable and semantic-aware features in modeling workbenches [1, 16]. For example, dynamic filtering is a type of interactive features that permits filtering out model elements to focus on a specific concern [11, 39, 41, 64, 66, 77]. In [11], the authors developed dynamic filters to temporary display the (sub-)inheritance tree of a selected class of a metamodel, in a class-based metamodel viewer. Such features are semantic-aware, as they rely on concepts of the DSML under study (here, the inheritance concept defined in the involved metamodel). However, building such interactive features for each graphical modeling workbench by hand is a complex and time-consuming job [11, 50]. So, there is a scientific challenge in helping language designers in building graphical modeling workbenches that embed interactive features specifically adapted to modeling and proposed in the literature. To overcome this challenge, we identified the following research questions that currently prevent the build of such graphical modeling workbenches:

(1) How to permit language designers to add interactive features dedicated to modeling in their different modeling workbenches? If the scientific literature proposed various interactive features, the current DSML approaches do not propose the integration of such features in modeling workbenches.
(2) How can a language designer configure selected interactive features to match the DSML concepts? Each DSML has its own concepts. So, configuring a semantic zoom, for example, requires to map the different zooming levels to elements of the targeted DSML (*e.g.,* elements of its abstract or concrete syntaxes).
(3) How to integrate the two above-mentioned points in the model-based development process of a DSML and its modeling workbench? The classical DSML development approach involves the definition of different metamodels to build one DSML. Each metamodel describes one aspect of the DSML (*e.g.,* its concrete syntax, its abstract syntax). A language workbench then integrates the different metamodels to build as output a modeling workbench.

This paper proposes a novel approach to help language designers in building modeling workbenches with dedicated semantic-aware interactive features. In addition to the existing meta-languages that define a DSML (*e.g.,* meta-languages that define the abstract and the concrete syntaxes), the approach proposes a novel meta-language to select and specialize the interactive features that language designers want in the modeling workbench of a given DSML. By extending a classical DSML approach, our proposal complements existing language workbenches dedicated to the production of graphical modeling workbenches.

We implemented our proposal within an industrial open-source language workbench, namely Sirus Web [35]. Our evaluation, based on the development of two representative graphical DSMLs, supports discussion of the feasibility of the approach. We also evaluate the scalability of it.

To sum up, the contribution of the paper consists of a novel approach to complement existing language workbenches by describing and specializing semantic-aware interactive features of graphical modeling workbenches.

99 The rest of the paper is organized as follows: Section 3 presents the challenges of easing the
100 production of semantic-aware interactive features, and a review of common interactive features.
101 Section 4 details the proposed approach. Section 6 describes an implementation of our approach in
102 a language workbench. Section 6 presents the evaluation of the proposal. Section 7 comments the
103 related work. Section 8 concludes the paper and discusses future work.
104
105 ## 2 BACKGROUND AND ILLUSTRATIVE EXAMPLE
106 This section first introduces the classical model-driven engineering approach used to develop
107 DSMLs. It then details an illustrative example used throughout the paper to both motivate the
108 scientific challenges and the proposed approach.
109
110 ### 2.1 Background
111-129



130 Fig. 1. The model-driven development approach of DSMLs.
131
132 Figure 1 depicts the standard model-driven approach to develop DSMLs [21]. This approach
133 involves two roles. The *language designers* are in charge of developing the DSML and its development
134 environment (the *modeling workbench*). To do so, they use a specific development environment,
135 called a *language workbench*, such as XText for textual languages [26], or Sirius Web for graphical
136 languages [35]. The interested reader can refer to [28] that discusses a large panel of language
137 workbenches and their characteristics. In a language workbench, a language designer creates a
138 DSML by designing the different metamodels that compose it, in particular: the abstract syntax
139 metamodel that details the concepts of the DSML; the concrete syntax metamodel that defines how
140 to represent graphically or textually, the concepts (a DSML can have several concrete syntaxes);
141 other metamodels that define specific concerns of the DSML and its modeling workbench (*e.g.*,
142 a language designer can define the debugging metamodel of an executable DSML [14]). Each
143 of these metamodels focuses on one specific concern of the DSML. The language workbench
144 combines and integrates these metamodels to produce a modeling workbench. One can extend a
145 DSML to add, for example, new features in the modeling workbench or a new concrete syntax.
146 To do so, one has to: ❶ – define a novel type of metamodels (a meta-metamodel), such as the
147

interactivity meta-metamodel proposed in this paper; ❷ – extend the integration to consider the novel meta-metamodel. For example, when adding debugging facilities to a DSML, developers of this debugging approach have to code how one designed debugging metamodel can augment the output modeling workbench with the defined debugging facilities. *Modelers* use the modeling workbench produced to edit models of the involved DSML. A model instantiates elements of the abstract and concrete syntax metamodels: the abstract syntax of a model conforms to its metamodel.

## 2.2 Illustrative Example

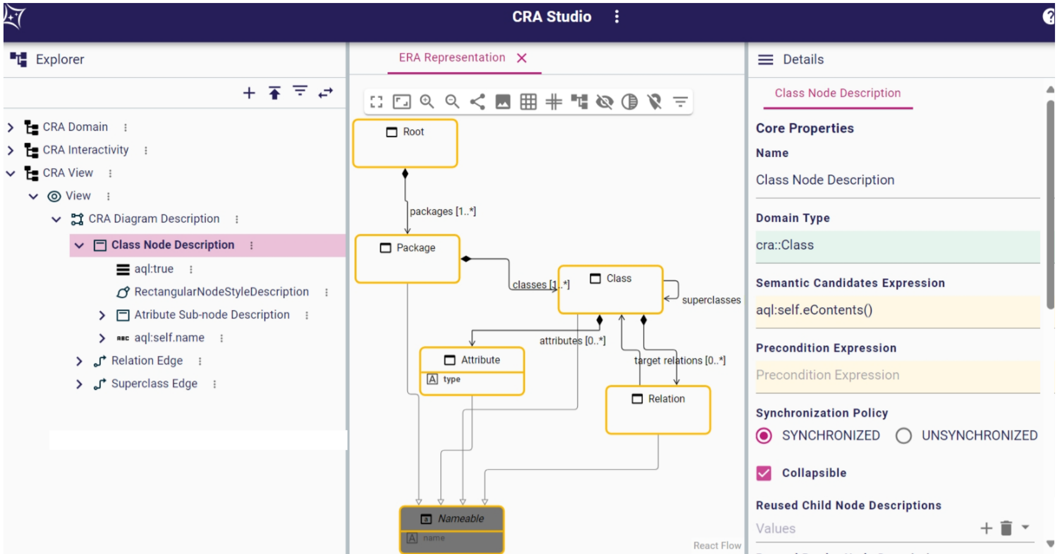



Fig. 2. A language workbench used to defined the abstract and concrete syntaxes of the CRA DSML

We will use a simple DSML to illustrate the standard model-driven DSML development approach, and explain the scientific challenges and the proposed approach detailed in the next sections. We named this illustrative DSML *CRA* (*Class Relation Attribute*), as it is a simple UML-like class diagram DSML. Following the process depicted by Figure 1, a language designer uses a language workbench (here Sirius Web [35]) to build a modeling workbench for the CRA DSML. The abstract syntax metamodel of CRA is first designed (middle pane of Figure 2). The `Root` class is the root of the abstract syntax metamodel and can contain `Package` instances which, in turn, can contain `Class` instances. A `Class` instance can be abstract and may have a superclass (of type `Class`). It can also contain multiple `Attribute` and `Relation` instances. A `Relation` (that stands between two classes) has a unique `Class` target. Finally, `Package`, `Class`, `Attribute`, and `Relation` inherit from the interface `Nameable`.

The language designer can now create a graphical concrete syntax metamodel that will represent the concepts of the abstract syntax metamodel. The left pane of Figure 2 partly depicts a concrete syntax for CRA. A *node description* corresponds to the graphical representation of a class of the abstract syntax metamodel (here the class `cra::Class`). The style description defines the style of this graphical node. The two *Edge* items describe the graphical representation of two types of links between classes: inheritance (superclass) and relation.

The language workbench then uses the abstract and concrete syntax metamodels to build as output the modeling workbench. This modeling workbench will embed standard interactive features,

197 such as a physical zoom, or a tool palette, to create model elements. Even if the abstract and concrete
198 syntaxes of this DSML are simple, the use of this DSML can lead to large models with numerous
199 classes and relations [11]. In such cases, a user may face difficulties in understanding, editing, or
200 navigating within the models, for example, to find a specific class or visualize the connected classes.
201

## 3 CHALLENGES

202
203 The DSML development example introduced in the previous section would lead to a modeling
204 workbench with standard interactive features. A reason is that interactive features specifically
205 dedicated to each DSML require specific developments that current DSML development approaches
206 do not propose. The literature, however, details a large panel of such dedicated interactive features,
207 which we identify and discuss in Section 3.1. Section 3.2 then discusses the challenges that language
208 workbenches face for proposing such interactive features.
209

### 3.1 Semantic-aware interactive features

210
211 This subsection details a set of common interactive features that aim at improving usability and
212 productivity while editing a graphical model in a modeling workbench. We focused on semantic-
213 aware interactive features, as defined as follows:
214
215 *Definition 3.1 (semantic-aware interactive feature).* A semantic-aware interactive feature of a
216 modeling workbench requires user interactions to perform some actions on a view of the model
217 (*e.g.,* navigation actions) or on the model itself (*e.g.,* editing actions). Those features are aware of the
218 context and configured according to the DSML syntaxes. By context we mean the current activity
219 of a modeler, for example working on a given model element.
220
On the contrary, generic interactive features are reused among languages, such as the physical
221 zoom (in [61], the authors called such features diagram-aware features). Another example is the
222 feature *folding* that folds the children of a given node. This feature is generic, as it applies to any
223 parent-child pattern.
224 Since there is no specific term to identify interactive features of graphical modeling environ-
225 ments, we did not use a systematic approach. Instead, we reviewed research papers published in
226 specific venues, both in the Human-Computer Interaction (VL/HCC, CHI, UIST, EICS) and software
227 engineering (SoSyM, MoDELS, SLE, VisSoft, CoLa) domains in the past five years. To do so, we
228 analyzed their proceedings using DBLP[1] by reading their titles and abstracts. We manually selected
229 papers that specifically refer to both software modeling and HCI. We also consider the references
230 from a taxonomy paper [13]. This provides a selection of related work papers and papers discussing
231 interactive features for graphical modeling workbenches. We also processed the references of the
232 selected papers according to a snowballing approach. We completed this set with scientific papers
233 from our personal bibliography. The goal of this step is not to be exhaustive, but to identify a
234 representative set of interactive features.
235 Table 1 lists the 16 interactive features we identified. In this table, we did not include interactive
236 features that perform CRUD operations since those features natively come with any modeling
237 workbench. From this set, 15 come from our review of specific venues. The last remaining one
238 comes from our personal bibliography (a Petri net modeling workbench [6]). During the process,
239 we also identified six research papers we discuss as related or motivating work [24, 32, 37, 48, 54].
240 The identified research work proposes either novel features for interacting with graphical models
241 (*e.g.,* semantic zooms), or services to be plugged into modeling workbenches through interactive
242 features (*e.g.,* recommender systems).
243

244 [1]For example with EICS and MoDELS: https://dblp.org/db/conf/eics/index.html, https://dblp.org/db/conf/models/index.html.
245

Table 1. Identified interactive features and services.

| **Name** | **Categories** | **Sources** | Supported |
|---|---|---|---|
| Semantic Zooming | Visualization | [11, 22, 30, 36] | ✓ |
| Magic lens | Visualization | [59, 60] | |
| Dynamic Filtering | Visualization | [11, 39, 41, 64, 66, 77] | ✓ |
| Focus+context | Visualization | [41, 58] | |
| Offscreen | Visualization | [22, 29] | |
| Hover | Visualization | [61] | |
| Edge navigation | Navigation | [11, 76] | |
| Semantic search | Navigation | [5, 11, 76, 77] | ✓ |
| Auto-layout | Editing | [61] | |
| Quick-fix | Editing | [27] | |
| Graphical auto-completion | Editing | [6, 47] | ✓ |
| Template | Editing | [46, 72, 74] | ≈ |
| Stroke gesture | Editing | [31, 65] | |
| Recommender system | Editing | [3, 23, 75] | |
| Model assistant | Editing | [18, 52, 68] | |

Our work focuses on the use of mice and keyboards so that we let stroke gestures as future work. Moreover, we focus on interactive features that cannot be automatically inferred from the different metamodels of the DSML (*i.e.,* generic interactive features). For this reason, we excluded edge navigation or off-screen visualization, which can be inferred from the concrete syntax. Some features, such as the basic auto-completion, can be inferred from the concrete syntax definition. For example, in a 2D context, a graphical auto-completion can consist in completing an element under creation with its mandatory elements, inferred from the syntaxes [6]. An auto-completion system, however, can also be semantically aware. For example, graphical auto-completion can suggest model snippets while editing. Services like recommender systems, model assistants and quick fix rather work as an external service to build and plug-in. We thus do not consider such features. For the same reason, we did not consider automatic graphical layout algorithms, as they usually do not require user interactions.

In this paper, we focus on four semantic-aware interactive features. A *semantic zoom* applies filters depending on the zoom level [11, 22, 30, 36]. A filter performs operations (hide, show or modify graphical elements) on the current view. A semantic zoom is useful to visualize a model with different levels of details, where a modeler can change the level of details using, for example, zooming in/out. The *dynamic filter* enables filtering out elements of a view based on a selected element and the related information to highlight [11, 39, 41, 64, 66, 77]. For instance with a class diagram, a dynamic filter can display the super inheritance tree of a selected class and hide the rest of the view. The *semantic search* is a search field that searches in specific configured parts of models [5, 11, 76, 77]. The templating interactive feature consists in proposing model fragments to be merged and completed in the current model under edition. This principle is partly shared

with the graphical auto-completion. We call this share principle *snippet*, our approach supports. A snippet is simpler than a template, as a template can have parameters. We do not consider the magic lens and the focus+context techniques, as they rely on concepts similar to semantic zoom and dynamic filter (*i.e.,* hide, show or modify graphical elements).

### 3.2 Categories of gaps between specifying and producing interactive features of graphical modeling workbenches

Adding graphical semantic-aware interactive features in modeling workbenches, such as the ones detailed in the previous section, requires overcoming several scientific and technical gaps. These gaps currently prevent language designers to build graphical modeling workbenches with semantic-aware interactive features.

*Semantic gap.* One specificity of graphical DSMLs and their modeling workbenches is that a large panel of interactive features is expected to be configured based on the semantics of the DSML under study (see Table 1). For example, adding a semantic zoom requires identifying which elements of the CRA DSML to hide, show, or transform, on each zooming level. Such elements can be part of the different metamodels of the DSML, such as the abstract syntax, the concrete syntax, or the mapping between these two syntaxes. Therefore, specifying the interactive features of graphical modeling workbenches requires access to the different metamodels of the DSML, such as the ones depicted by Figure 2 for the CRA DSML.

*Performance gap.* The execution of interactive features may be expensive on large models. For example, with cloud-native workbenches (*i.e.,* Web applications composed of back-ends that provide services and front-ends that provide user interfaces and use those back-end services), interactive features may require back-end queries. This may have a negative impact on the usability of the modeling workbench by increasing the latency [61]. For example, a semantic zoom may have to filter in or out numerous model elements and may require supplementary information from a back-end.

*Platform gap.* The transformation from the DSML specifications (*i.e.,* the different metamodels, step ❷ in Figure 1) to a concrete modeling workbench targets platforms that have their own technical constraints and interactive features. For example, targeting the Eclipse or the Sirius Web platforms requires dealing with their UI frameworks and existing features (*e.g.,* physical zooms). Another example is that interactive features may either result in the creation of a new view or a modification, temporary or permanently, of the current view. The targeted platform may impose such a design choice.

*UI gap.* Adding interactive features may require the addition of new widgets in the UI of the modeling workbench. To do so, the approach must have access to information related to the UI of this workbench.

> Increasing the interactivity of graphical modeling workbenches of DSMLs faces several challenges (the *gaps*) we have detailed in this subsection. The proposed approach is the result of a trade-off among these gaps. The next subsections detail our proposals for bridging these gaps.

# 4 APPROACH

This section details the proposed approach. Section 2.2 introduces an illustrative example. Section 4.1 gives an overview of the approach. Section 4.2 details the proposed meta-language used to specify semantic-aware interactive features to be used in modeling workbenches.

## 4.1 Overview

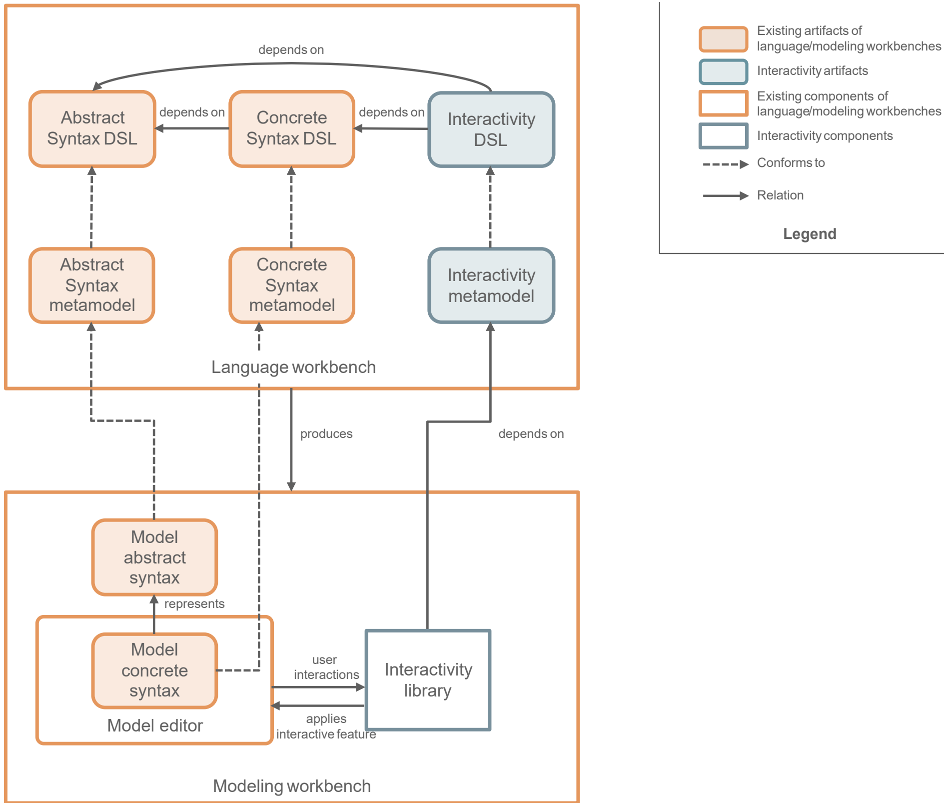



Fig. 3. Overview of the approach

Figure 3 presents the overview of the approach that overcomes the identified gaps (Section 3.2). To overcome the *semantic gap*, we propose a new meta-language (see Figure 1) that permits language designers to select and customize semantic-aware interactive features for a given modeling workbench. This new meta-language, namely **Interactivity meta-language** (that produces **interactivity metamodels**), complements the mainstream MDE development approach used to build DSMLs and their modeling workbenches (detailed in Section 2): in addition to other metamodels (concrete syntax, abstract syntax, *etc.*) language designers create to build their modeling workbenches, they can now use the *Interactivity* meta-language we propose to customize the interactivity of modeling workbenches. Section 4.2 introduces this new meta-language.

To overcome the *platform gap*, the *Interactivity* meta-language is independent of the targeted language workbench (*e.g.*, Sirius Web). To concretize an interactivity metamodel to a given language workbench, each modeling workbench has its own **interactivity library**. A developer has to code

one interactivity library for a given language workbench. The interactivity library extends the produced modeling workbenches with the interactive features our proposal supports. It interprets the interactivity metamodel developed by the language designer to configure the modeling workbench using, for example, the modeling workbench API. This enables widget addition (*e.g.*, buttons) and event listening (*UI gap*). The *Interactivity* library then reacts to the different listened user interactions to apply the interactive features.

### 4.2 Specifying semantic-aware interactive features

Using the proposed *Interactivity* meta-language, language designers can select the interactive features to include in their modeling workbench. Then, they have to configure the selected features to adapt them to the DSML under development. Figure 4 depicts the abstract syntax of the *Interactivity* meta-language. Listing 1 is an illustrative metamodel produced using the *Interactivity* meta-language for the CRA DSML (introduced in Section 2.2).

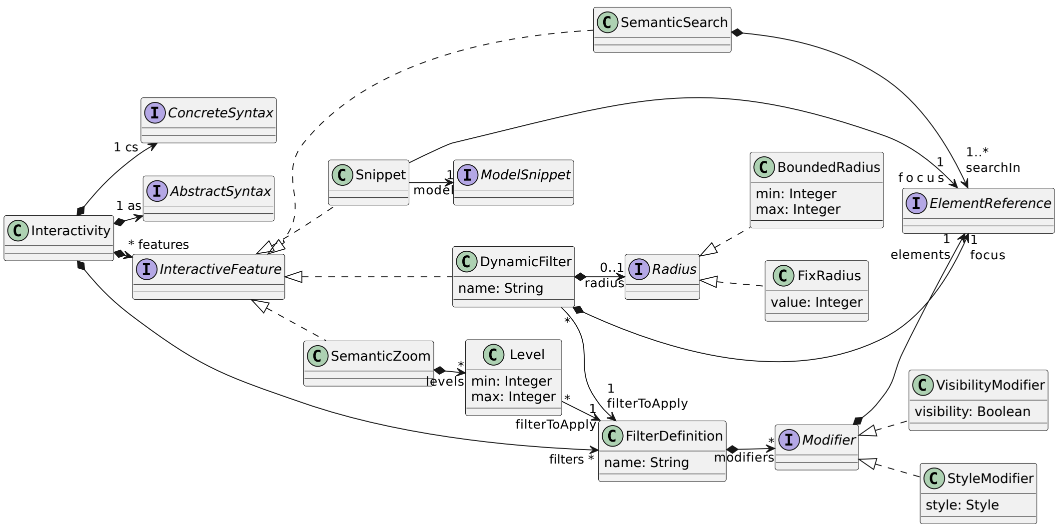



Fig. 4. The meta-metamodel of the *Interactivity* meta-language

The *Interactivity* abstract syntax makes references to the other metamodels of the DSML (*e.g.*, abstract and concrete syntaxes). In Figure 4, the class `Interactivity` has relations to the handled concrete and abstract syntaxes. In Listing 1, lines 1 and 2 indicate the abstract and concrete syntaxes of the CRA DSML. Then, a language designer can select different interactive features identified and introduced in Section 3.1: the semantic search (class `SemanticSearch`, and illustrated on Lines 3 to 5); the dynamic filtering (class `DynamicFilter`, and illustrated from lines 6 to 10); the semantic zoom (class `SemanticZoom`, and illustrated from lines 11 to 13); the snippet-based features (class `Snippet`, and illustrated from lines 14 to 19).

Once selected, the language designer must specialize a semantic-aware interactive feature to match the DSML. The semantic search must select the elements from the DSML to search. For example, on line 4, the search results will first display matching class names, then package names. If the language designer sets `all` to `true` (Line 5), then the search result list will end with all the remaining string occurrences found in the DSML. In the meta-metamodel, we represent references to DSML elements through the interface `ElementReference` to let the language workbench developers

choose how to implement this one. In Listing 1, a string represents such a reference, such as *craas.packages.classes.name* on line 4.

The dynamic filters and the semantic zoom are visualization techniques based on the concept of filters that temporarily change the view. In the `filters` section (from Line 21), the language designer defines the filters (class `FilterDefinition`) to be used in those interactive features. In the code example, the language designer defined three filters: *show-inheritance* (Line 22), *without-attributes* (Line 25), *without-package* (Line 29). A filter has a set of modifiers that show, hide, or change the style of targeted elements of the abstract syntax (`Modifier.elements`). For example with Line 24, the modifier will show the target objects of *package.class.superclasses.* The default behavior of a filter definition (*e.g.,* show by default all the elements, while the filter will hide some of them) depends on the interactive feature that uses it, as detailed below.

A modeling workbench can have several dynamic filters. Our code example defines a single one (Line 6) named *inheritance* (Lines 7 to 10). This filter shows the super inheritance tree of a targeted class. This dynamic filter first defines the element of the abstract syntax from which the filtering starts, here `craas.packages.classes` (Line 8, the *focus*). The selected filter is *show-inheritance* (Line 10), defined on lines 22 to 24. The default behavior of a filter in the context of a dynamic filter consists in hiding all the elements. Then, the applied filter shows or changes the style of selected elements recursively: the first 'show' modifier (Line 23) means that `Class` objects contained in packages will be displayed. From the focused class, the dynamic filter will apply the filter to each element that is the target of the `superclasses` relation (Line 24), recursively until reaching the radius (Line 9). The radius is an integer value that permits limiting the propagation of the dynamic filter, *i.e.,* the number of elements displayed around the targeted element. In the context of the *inheritance* dynamic filter, a radius of 0 would mean that only the targeted class and its super classes will be displayed, without any propagation to the super classes.

The semantic zoom does not focus on a given element but evaluates its filters globally (*i.e.,* from the roots of the model). By default, all the abstract syntax model elements are considered visible unless a modifier indicates otherwise. A semantic zoom has different zooming levels (Lines 12 and 13). To each level corresponds a filter configuration. For example, on line 12 the *without-attributes* filter is executed when the zoom level is between 0% and 75%. This level hides class attributes (Line 26) and increases the font size of the class names (Line 27). Between 150% and 200%, the applied filter is *without-packages* (Line 13). This filter hides the packages (Lines 28 and 29) but not the classes they contain.

Finally, the snippet interactive feature involves a model snippet (interface `ModelSnippet`). A model snippet corresponds to a model fragment to be inserted into the current model. It thus relies on the abstract syntax of the DSML under development. `ModelSnippet` is an interface that abstracts snippet specification. The developer can develop the most suitable concrete implementation of `ModelSnippet` for a giving language workbench. For example, one may choose to specify the snippets as a model which is an instance of the developed DSML if the language workbench enables its instantiation at design time. Another one may rely on the ESON DSML [2] that permits the creation of models in a textual format. The example of Listing 1 defines two snippets: *abstractClass* (Line 15); *composite* (Line 18). The *abstractClass* snippet corresponds to the definition of an abstract class (*i.e.,* an object of type `Class` with the attribute `abstract` set to *true*). The *composite* snippet is more complex and corresponds to the design pattern *composite* (a class $X$ that inherits from class $Y$ and is composed of objects of type $Y$) [34]. Both snippets have a focus element, here `craas.packages`, that defines the merge point of the snippet into the model. Therefore, the snippets may be proposed to the modeler when they interact with an instance of the focus element. In the example, when

[2] https://wiki.eclipse.org/ESON

a modeler selects a package, the user interface may suggest the two snippets. Note that in the *Interactivity* DSL, the snippet feature is independent of the concrete interactive features that would trigger the snippet in the modeling workbench. For example, a model snippet can take the form of a button in an editing palette (so a kind of templates), or be the result of a drag-and-drop (*i.e.,* the auto-completion drag-and-drop of [6]). Future work may consider templating (*i.e.,* snippet with advanced features, such a template parameters [46]). The snippet interactive feature can also specify an element of the DSML concerned by the snippet.

The *Interactivity* DSL configures interactive features without considering how they are triggered inside the modeling workbench. Indeed, triggering a user interaction depends on the modeling workbench user interface. So, it depends on the modeling workbench capability and its tech stack. Given the diversity of language workbenches and tech stacks, the way of triggering user interactions is left up to the developer.

```
1 import abstract-syntax 'cra-as.ecore' as craas
2 import concrete-syntax 'cra-cs.ecore' as cracs
3 search:
4   in: [craas.packages.classes.name, craas.packages.name]
5   all: true
6 dynamic-filter:
7   inheritance:
8     focus: craas.packages.classes
9     radius: [1..*]
10    filter: show-inheritance
11 semantic-zoom:
12     [0%-75%[: filter: without-attributes
13     [150%-200%[: filter: without-packages
14 snippet:
15   abstractClass:
16     model: absclass.eson
17     focus: craas.packages
18   composite:
19     model: composite.eson
20     focus: craas.packages
21 filters:
22   show-inheritance:
23     show: craas.packages.classes
24     show: craas.package.class.superclasses
25   without-attributes:
26     hide: craas.packages.classes.attributes
27     setstyle craas.packages.classes.name: font-size 200%
28   without-package:
29     hide: craas.packages
```

Listing 1. An interactivity metamodel for the CRA DSML

540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588

### 4.3 Composability of semantic-aware interactive features

Our approach supports the activation of several interactive features simultaneously. For example, a modeler can zoom in semantically so that some elements are not displayed anymore. Then, this user can apply a dynamic filtering. Such compositions of features imply conflicts (see Table 2) that need strategies to be resolved.

Table 2. Conflicts between interactive features. ✓ means that applying the feature 1 and then the feature 2 leads to a conflict. ✕ means that no conflict appears.

| Feature 1 \ Feature 2 | Dyn. filter | Sem. zoom | Sem. search | Snippet |
|---|---|---|---|---|
| **Dyn. filter** | ✓ | ✓ | ✓ | ✓ |
| **Sem. zoom** | ✓ | ✓ | ✓ | ✓ |
| **Sem. search** | ✕ | ✕ | ✕ | ✕ |
| **Snippet** | ✕ | ✕ | ✕ | ✕ |

The use of visualization features (dynamic filter and semantic zoom) implies conflicts, whatever the feature that is then used. Two successive dynamic filters result in applying the second one (and replacing the first one): dynamic filters are not cumulative, including in scenarios where features are executed between the two filters. The same scenario operates with two semantic zooms. A dynamic filter followed by a semantic zoom is equivalent to performing the semantic zoom followed by the filter (idempotence). The idea is that semantic zooming prevails over dynamic filtering. Therefore, this also means that a dynamic filter cannot display elements that the semantic zoom hides. A semantic search applied after a dynamic filter or a semantic zoom searches in the displayed elements only. Similarly, a snippet cannot be used if it contains an element that a dynamic filter or a semantic zoom currently hides. The use of a semantic search or a snippet followed by any interactive feature does not lead to any conflict.

## 5 IMPLEMENTATION

This section presents an implementation of the *Interactivity* meta-language. Our implementation operates within the graphical language workbench Sirius Web [35]. Section 5.1 first presents Sirius Web, and then Section 5.2 presents our implementation.

### 5.1 Sirius Web

We implemented our proposal within the industrial language workbench Sirius Web [35] version 2024.5.6. Sirius Web is an open-source project maintained by the Obeo company and hosted by the Eclipse Foundation [35], licensed under EPL v2. Sirius Web is a web application that final users can access through a web browser front-end. The Sirius Web's client (front-end) is developed with TypeScript, React, and *ReactFlow*[3] for the diagram rendering. Sirius Web uses *GraphQL* to exchange data between the front-end and the back-end. Finally, the server part (back-end) is developed using Java, Spring Boot, and EMF [69] (Eclipse Modeling Framework). EMF is an industrial framework that supports the development of meta-metamodels through the Ecore meta-language (a meta-language used to define meta-languages) [69]. Numerous language workbenches use EMF and Ecore as a core framework to define their meta-languages. Sirius Web is one of them, providing language designers with two meta-languages made with EMF: *Domain* to define the abstract syntax of the

[3] https://reactflow.dev

DSML; *View* to define the concrete syntax and the tools enabling, from the concrete syntax, the creation, modification, or deletion of abstract syntax model elements. The View DSL enables the development of various kinds of concrete syntax: diagram, form, Kanban, and Gantt. To map the abstract syntax to the concrete syntaxes, Sirius Web develops and uses a query language called *AQL* (*Acceleo Query Language*). Thus, a specific concrete syntax node or edge may correspond to zero, one, or multiple abstract syntax elements or relations. The developed languages are automatically deployed and then interpreted inside Sirius Web itself. The different (meta-)languages are usable inside projects. Sirius Web proposes two kinds of projects: the *studio* projects where a language designer can use the two meta-languages; and the *basic* projects where a modeler can use the different developed DSMLs. Finally, Sirius Web is collaborative. It can manage multiple clients at the same time, editing a shared representation. When a modeler modifies the model, the concrete syntax is recomputed on the server and the modifications are broadcast to all the clients who visualize it.

### 5.2 Interactivity library architecture

As a reminder, supporting a language workbench requires implementing an interactivity library for it (see Figure 3). As our current implementation operates within Sirius Web, our interactivity library implementation is developed using the same technology stack. The implementation within Sirius Web is available on Github[4]. The abstract syntax meta-metamodel of the *Interactivity* DSL is developed using *Ecore*, which permits the code generation of the meta-metamodel. This generated code directly works within Sirius Web to enable the use of the meta-language inside the Sirius Web DSML projects. Since Sirius Web has its query language, *AQL*, we decided to use it to represent the references of the *Interactivity* DSL to the developed meta-language abstract syntax elements. As concrete syntax, we use the default reflective tree editor Sirius Web provides for all the (meta-)languages. The lifecycle (creation, persistence, and deletion) of metamodels based on the *Interactivity* DSL is managed by Sirius Web similarly to other (meta-)languages.

Regarding the interactive features, a final user of a generated modeling workbench controls the semantic zoom using the physical zoom (*e.g.,* by zooming in/out using a scroll). For the dynamic filter and the snippet, we decided to add buttons inside the contextual palette of the associated concrete syntax elements to trigger the interactive features. For instance, based on the CRA DSML and its interactivity metamodel example (see Section 2.2), all the concrete syntax elements representing the classes will have a button inside their contextual palette to trigger the `inheritance` dynamic filter.

*5.2.1 Architecture.* Figure 5 illustrates the architecture of the Sirius Web with the *Interactivity* library. In this figure, only the interaction of the semantic zoom is represented for readability purposes.

An interactive feature is triggered by the modeler, on the client side. Our implementation operates similarly as processing user interface events [8, 10]: specialized interactive features trigger dedicated commands that change the state of the modeling workbench. To listen to user interactions, such as the physical zoom, we implemented the different interactive features as React components we placed as children of an `Interactivity` component, nested in the Sirius Web diagram renderer. Therefore, the diagram renderer can give the different interactive features the information they need, such as the interaction listeners or the concrete syntax (❶). The model concrete syntax comes from the server when the modeler opens a representation or when the model is changed while the representation is open (❷). To configure the different interactive features, the interactivity component queries the server to get the interactivity metamodel of the DSML used for the opened

[4] https://anonymous.4open.science/r/Interactivity-5031/README.adoc

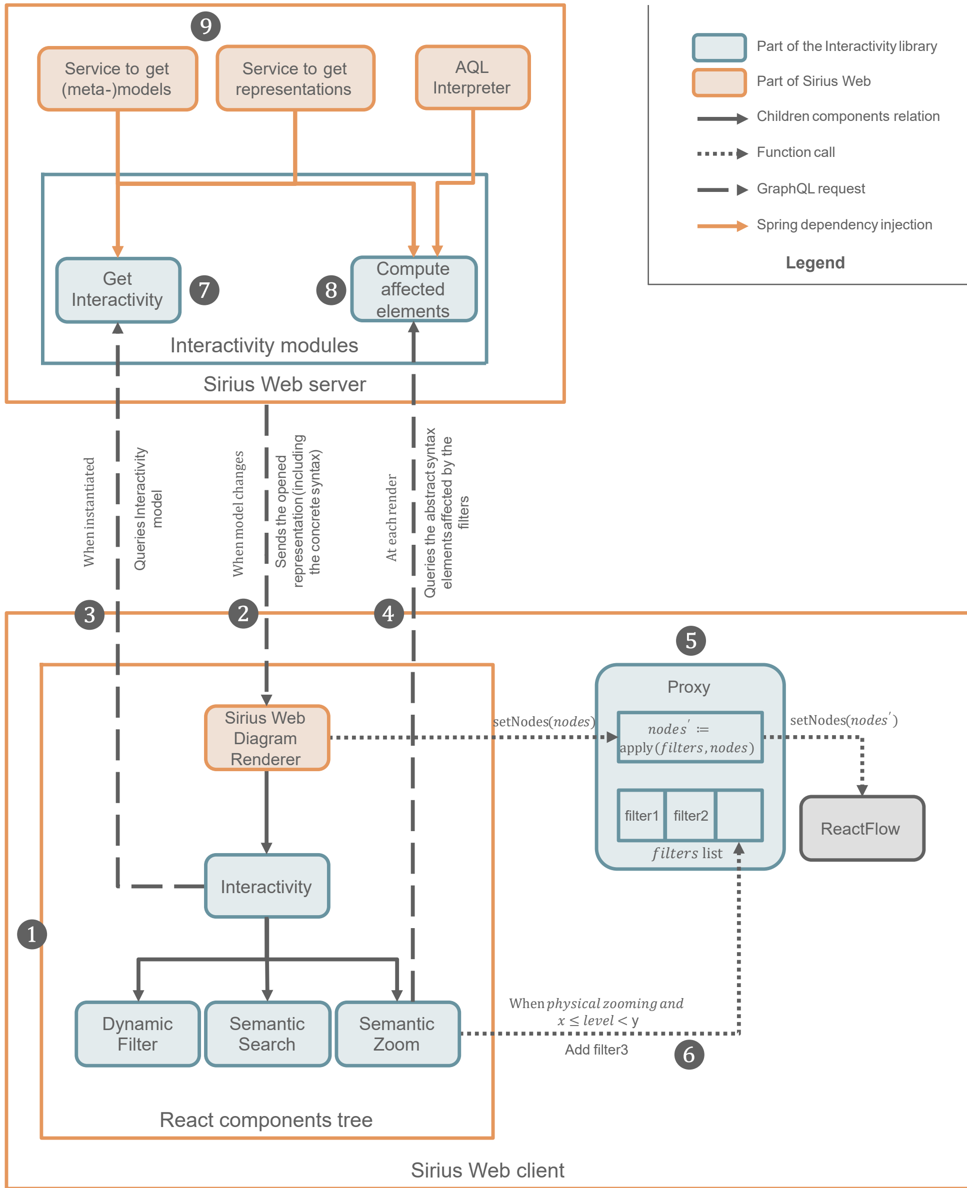



Fig. 5. Simplified architecture of the *Interactivity* library inside Sirius Web.

model (❸). Thanks to the interactivity metamodel, the `Interactivity` component can choose which interactive feature components must be instantiated, and configure them according to the metamodel.

In Sirius Web, the client does not have any information about the opened model except its representation (including the concrete syntax). To decide which concrete syntax elements to filter, we need to evaluate the filter over the abstract syntax on the server side. Therefore, at each new render of the representation in the client, the semantic zoom queries the identifiers of the abstract

syntax elements affected by the active filters (❹). Each concrete syntax node contains the identifier of the abstract syntax element it represents. When applying the filter over the concrete syntax, the client simply has to filter the concrete syntax nodes that contain the identifier returned by the server. To modify the concrete syntax, the filters should be applied before the rendering. In practice, the rendering of the concrete syntax is not directly managed by the Sirius Web diagram renderer but delegated to a third-party library named *ReactFlow*. In order to pass the nodes and edges to *ReactFlow*, *ReactFlow* proposes several functions such as `setNodes`. We proxied these functions to modify the concrete syntax before displaying it, for instance to apply the filters over the concrete syntax (❺). The active filters are stored by the proxy and applied each time Sirius Web calls the `setNodes` function. When the semantic zoom is triggered, the concerning filter is stored inside the proxy and removed when the associated semantic zoom level is disabled (❻).

On the server side, we developed new Spring modules, allowing us to benefit from the different Spring services that Sirius Web proposes. For instance, we can register our own *GraphQL* entry points to communicate with the client to get the interactivity metamodel (❼) or the abstract syntax elements' identifier affected by an interactive feature (❽). We can also access the services enabling us to obtain the different representations and (meta-)models, including the Interactivity, Domain, and View metamodels describing the used DSML (❾).

*5.2.2 Modification of Sirius Web.* Thanks to Spring, our implementation on the server requires no modification of the original Sirius Web source code, except to integrate it into the Maven build scripts. However the client of Sirius Web is not as extensible as the server. For instance, considering the Sirius Web version 2024.5.6, it is impossible to modify the concrete syntax before displaying it without changing the original code. Moreover, some required functions and types were not exposed by Sirius Web. For instance, the function that converts a concrete syntax node style to a type compatible with ReactFlow is not accessible outside the code of Sirius Web. Our main modifications consisted in exposing certain functions we needed, adding our implementation to the Sirius Web diagram renderer, and adding the icons to the different palettes. Our modification only concerned the diagram part of the Sirius Web client.

*5.2.3 Conflicts with overlapping interaction features.* The targeted language workbench may already have interactive features our approach proposes, such as a semantic zoom. In this case, this is up to the *Interactivity* library to decide how to integrate the semantic zooming our approach propose within the modeling workbench. This can be done by replacing the native one (if possible), or by using another user interaction to trigger it. We did not face this issue with Sirius Web.

*5.2.4 Interactive features implementation.* The semantic zooming algorithm applies the AQL queries of the different path modifiers over the model abstract syntax. Then, it sends the identifiers of the different queried elements to the client, linked to a specific identifier for the modifier to apply to this element. From the model's concrete syntax, the client can obtain the identifier of the abstract syntax element represented by a specific concrete syntax element. Thanks to this identifier and those received from the server, the client can know if a concrete syntax element should be affected and by which modifier.

Closely related, the dynamic filter nevertheless has a very different algorithm for the server part. Here, the different AQL queries for the path modifiers are applied from the root of the model and by considering the focused element. In other words, the selected elements are those that have a direct reference from the focused element and that are returned by the AQL query. The paths are applied recursively by changing the focus to the new selected elements, until the radius is reached or there is no new selected elements. However AQL is a generic query language, so an AQL query may contain complex operations which may not make the path to the elements explicit. For instance,

*root.eAllContents().attributes* will return all the attributes of a model no matter which is the path. To identify the path taken by a query, we prune the model to force the query to pass through the focused element and we test all the branches starting from this element. Thanks to that, we can identify the path taken to go from a focus element to the selected elements, enabling us to show the relevant relations. Nevertheless, this implementation is heavier, so the choice of AQL here was perhaps not the most relevant one. Finally, the identifiers of the selected elements and relations are sent to the client, which uses the same algorithm as the semantic zooming.

The semantic search has a very similar algorithm to the semantic zooming. It applies the different AQL queries in the order defined by the language designer until one returns at least one result. The identifiers of the results are sent to the client. Finally, the client highlights the concrete syntax elements representing the abstract syntax elements matching the received identifiers. To do that, it uses the same algorithm than the semantic zooming and the dynamic filtering, with a hardcoded style modifier to highlight.

Finally, regarding the snippet feature, the algorithm consists of merging the snippet with the model. We use a basic merge strategy that adds the root elements of the snippet into the first compatible attribute/relation of the focused element. For example with the *abstract class* and *composite* snippets of the CRA example, the root elements of the snippets are class objects, which are merged into the focused package, in the *classes*.

## 6 EVALUATION

The evaluation aims to discuss the following research questions:

- RQ1 – **Feasibility.** Can the approach be used on representative DSMLs? This RQ aims at discussing to what extent the proposal can be used beyond simple use cases.
- RQ2 – **Performance.** Does the approach scale in terms of performance? When an interactive feature operates, it usually requires new data (*e.g.,* new data to be displayed). This RQ aims at discussing to what extent the queries executed to compute data affect the usability of the interactive features. This is particularly a point to discuss when the modeling workbench is separated into several parts (front-end, back-end), which may imply network queries.

Section 6.1 focuses on RQ1, through two use cases. Section 6.2 concerns RQ2. This section provides empirical performance measures and related discussions. Finally, Section 6.3 discusses the threats to validity and the scope of the proposal.

### 6.1 Use cases

This section details the use of our implementation of two representative use cases. The first use case is the CRA DSML, we developed to illustrate the proposal throughout the paper (Section 2.2). This use case is representative of small DSMLs (*i.e.,* its abstract and concrete syntaxes contain few elements). The second use case improves an existing DSML, named SysML v2. SysML v2 is the successor to the SysML standard for complex systems engineering, introduced by the OMG (Object Management Group). The Obeo company implements this DSML and its modeling workbench we used in this use case. This Obeo SysML v2 modeling workbench is called SysON[5]. The goal of this use case is to show how our approach can be used seamlessly within existing (industrial) DSMLs. This use case is representative of complex and industrial DSMLs, as SyML is an industrial standard.

*6.1.1 CRA DSML.* For the first use case, we used our implementation to build a Sirius-based modeling workbench for the CRA DSML. To do so, we now use the interactivity metamodel

[5] https://mbse-syson.org

described in Listing 2. Figure 6 depicts an example of a CRA model we produced using the developed Sirius Web modeling workbench.

```
import abstract-syntax 'cra-as.ecore' as craas
import concrete-syntax 'cra-cs.ecore' as cracs
search:
  in: [aql:root.packages.classes.name, aql:root.attributes.name]
  all: true
dynamic-filter:
  superclasses:
    focus: aql:root.packages.classes
    filter: show-superclasses
semantic-zoom:
    [0%-75%[: filter: hide-attributes
snippet:
  abstract-class:
    model: snippets/absclass.eson
    focus: aql:root.packages
  composite:
    model: snippets/composite.eson
    focus: aql:root.packages
filters:
  show-superclasses:
    show: aql:root.packages.classes
    show: aql:root.package.class.superclasses
  hide-attributes:
    hide: aql:root.packages.classes.attributes
```

Listing 2. Interactivity metamodel for the CRA DSML use case

The interactivity metamodel defines the four interactive features used in our approach. For three of them, we recorded a video that illustrates how the interactive feature operates in the CRA modeling workbench. As described in Section 5.1, the abstract syntax element references are AQL expressions.

Figure 7 is an example of a semantic search applied over the CRA model. First, the modeler types `double` in the search field (❶). The semantic search first looks at the class names, the attribute names, and then at any other string elements of the model. Since `double` is not used as a class or attribute name, the semantic search finds this value as the type of the `attribute2` attribute (❷). It then colors this attribute in red (❸). Figure 8[6] shows the results of using the `show-superclasses` dynamic filter. The filter is accessible through a contextual palette of the current node (❶). Using this filter results in showing the inheritance graph of the focused element only (here, `Class A`). Figure 9[7] shows the semantic zoom applied with a zoom level of 75%. At this level, it hides all the attributes of the classes. Finally, Figure 10 shows a toolbar that suggests the two snippets when

[5] https://youtu.be/H7LL-9e2qww
[6] https://youtu.be/EIyeQYYJ9M8
[7] https://youtu.be/fZuHYIHmac0

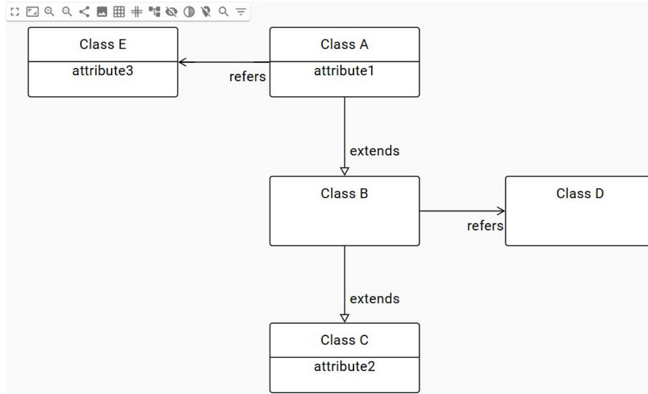



Fig. 6. Example of CRA model

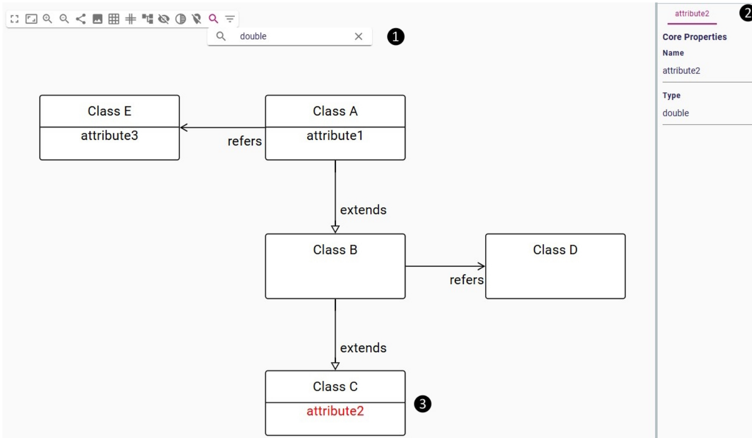



Fig. 7. The CRA model with the semantic search enabled[4]

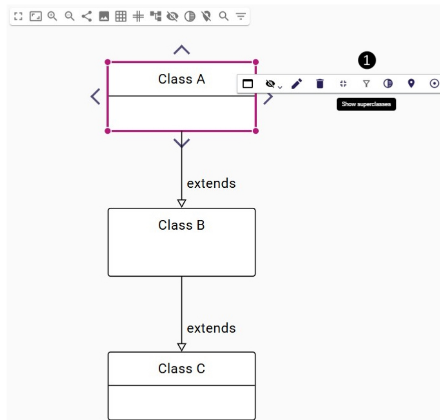



Fig. 8. The CRA model with the "superclasses" dynamic filter applied[5]

a package is selected. Clicking on one of these buttons will merge the snippet into the selected package.

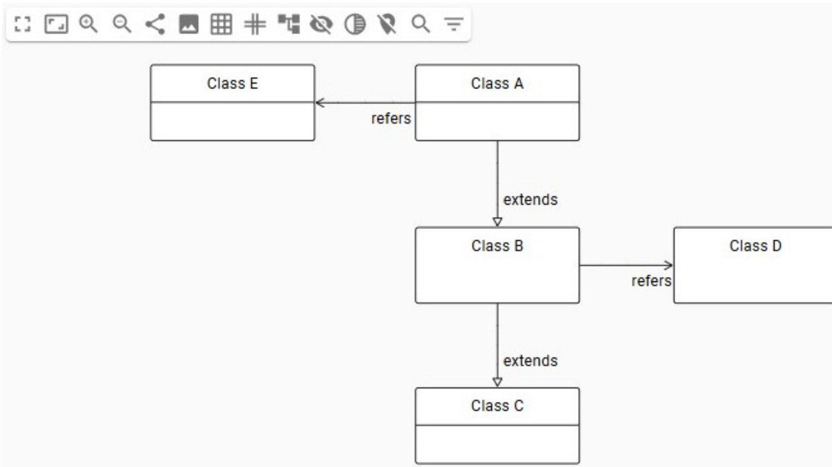



Fig. 9. The CRA model under 75%[6]

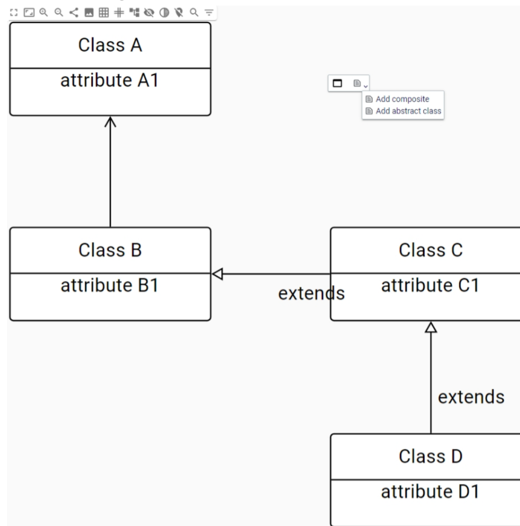


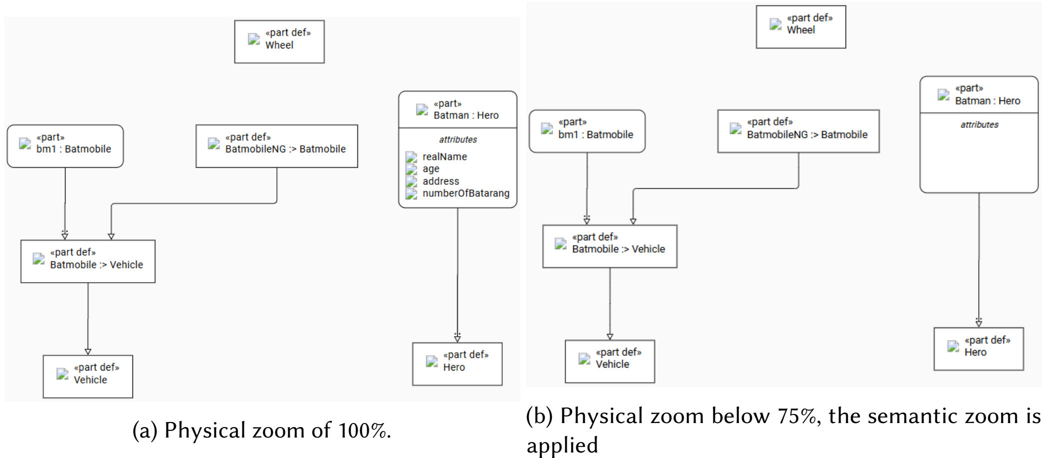

Fig. 10. The snippets buttons in the toolbar, using a CRA model

*6.1.2 SysML v2.* The second use case concerns the SysML v2 DSML. We implemented our use case by extending the SysON modeling workbench (version 2024.5.2). This version is based on Sirius Web version 2024.5.1.

```
1 import abstract-syntax 'sysml-v2.ecore' as sysml-v2
2 import concrete-syntax 'general-view.ecore' as general-view
3 search:
4   in: [aql:root.eAllContents()->filter(sysml-v2::Definition).name]
5 dynamic-filter:
```

```
6    only-reference-graph:
7      focus: aql:root.eAllContents()->filter(sysml-v2::PartUsage)
8      radius: [5]
9      filter: show-referenced-elements
10 semantic-zoom:
11   [0%-75%[:
12     filter: hide-attributes
13 filters:
14   hide-attributes:
15     hide: aql:root.eAllContents()->filter(sysml-v2::AttributeUsage)
16   show-referenced-elements:
17     show: aql:root.eAllContents()->eCrossReferences()
```

Listing 3. Interactivity metamodel built for the Sirius Web SysML v2 workbench.




(a) Physical zoom of 100%.

(b) Physical zoom below 75%, the semantic zoom is applied

Fig. 11. SysML v2 model describing Batman and his Batmobile in SysON enhanced with the interactivity library, at two different physical zoom levels

Figure 11a illustrates a SysML v2 model in SysON. In SysML v2, a `Definition` is the definition of a type of elements and a `Usage` is its usage in a certain context. A `Part` represents a system or a system component. An `Attribute` is a property of SysML v2 elements, like a `Part`.

Listing 3 describes the interactivity model we defined for SysML v2. The first two lines identify the abstract and concrete syntaxes on which the interactivity model will act. This model then defines the three interactive features. The semantic search looks for text in the name of `Path` objects only. The dynamic filter `only-reference-graph` shows `PartUsage` objects. The AQL query starts from the root of the model of the abstract syntax, gets all the children transitively (`eAllContents()`) and keeps only the instances of `PartUsage` (`filter(sysml-v2::PartUsage)`). The propagation is applied at most five times (the radius), starting from the focused element. Finally, the semantic

981 zoom filters the attributes (`AttributeUsage`) when the zoom level is below 75%. Figure 11 shows
982 the result before and after applying the semantic zoom.
983
984 **RQ1 – Feasibility. Can the approach be used on representative DSMLs?** We
985 implemented two use cases of different natures: we built the CRA DSML, a merely
986 simple DSML, from scratch; we extended the SysML v2 DSML, an industrial DSML,
987 to add interactive features. Both DSMLs have 2D graphical concrete syntaxes that
988 take the form of an entity-relation representation. Using our implementation, we
989 faced no issue to add the four types of interactive features. Our impact on the code
990 on the original Sirius Web language workbench to make the use cases operational is
991 minimal, and that for two main reasons: our approach seamlessly integrates a MDE de-
992 velopment approach; Sirius Web exposes development APIs to extend it with new features.
993
994
995 ## 6.2 Performance

996 We now focus on RQ2 related to the scalability of our implementation in terms of performance.
997 Discussing the scalability does not concern the meta-language we propose, but its implementation.
998 The goal is to observe whether a language workbench (Sirius Web in our case) can support semantic-
999 aware interactive features that require data to be computed within a short timeframe to be usable.
1000
1001 *6.2.1 Protocol.* To discuss scalability, we use a specific DSML called *Papaya* with three models of
1002 representative size. We did not use the DSMLs involved in the use case for the following reasons:
1003 CRA is a toy example and we do not have medium and large CRA models; SysML v2 is an industrial
1004 DSML with large models, but those models we have are not compatible with open science (they
1005 belong to companies). So, we use the *Papaya* DSML that is developed by the Obeo company for
1006 the specific purpose of scalability measures. *Papaya* models of different sizes are freely available.
1007 The Github companion repository contains the *Papaya* abstract syntax. In short, *Papaya* is closely
1008 similar to the CRA DSML, as it aims to define object-oriented architecture. Its abstract syntax
1009 metamodel contains 36 classes and 117 relations. The *Papaya interactivity* model is similar to the
1010 CRA one, except with semantic zooming, which hides operations in addition to attributes. The
1011 semantic search also searches first in attribute names then class names to obtain more results.
1012 We use three *Papaya* models to conduct the experiment: one *small* model composed of 246
1013 elements; one *medium* model composed of 6,200 elements; one *large* model that contains 47,297
1014 elements. The three models are available in the Github repository companion. Sirius Web comes
1015 with the large model, automatically computed from the Sirius Web architecture. We then simplified
1016 it to obtain a medium and a small models.
1017 We deployed Sirius Web and the Papaya modeling workbench on a remote server, outside the
1018 local network. We start measuring by executing the *Papaya* modeling workbench with one of
1019 the three models. We then execute one of our interactive features ten times for each model and
1020 each interactive feature. We then log the means computed from these ten executions. We have
1021 instrumented the Papaya modeling workbench to record needed performance data. In particular,
1022 we measure:
1023 - the execution time of the computation on the back-end (in ms);
1024 - the application time of the interactive features on the front-end (in ms);
1025 - the time to retrieve the result of the query for the interactive feature from the back-end, on
1026 the front-end (in ms);
1027 - the size of the network query result that contains the data that the back-end computed (in
1028 number of model elements).
1029

The third item includes the execution time of the interactive feature in the back-end, but also the network time (connection time, request/response time), and the time Sirius Web takes to dispatch the query to the interactive feature handler (and vice versa). We apply this protocol for three of the four interactive features: semantic zoom, dynamic filter, and semantic search. We do not consider the snippet feature, as it does not require much computation to work. We kept both semantic zooming and dynamic filtering, as their filtering algorithms differ.

The back-end measures are done using *System.nanoTime()* function from Java, and the front-end measures are done using *performance.now()* function from the JavaScript Performance API. Concerning the measure of the queries, we used the development tools of the browser.

Table 3. Execution times (in ms). Columns correspond to the average time measured over ten executions, with the standard deviation. Rows correspond to the interactive features according to the model size.

| Feature | Model size | Server time (ms) | Query time (ms) | Client time (ms) | Sent elements |
|---|---|---|---|---|---|
| **Semantic zooming** | **Small** | 31 (±7.93) | 65.3 (±7.36) | 1.76 (±0.77) | 65 |
| | **Medium** | 144.3 (±23.4) | 177.5 (±24.39) | 20 (±4.68) | 870 |
| | **Large** | 235.2 (±37) | 701.1 (±326) | 58.4 (±29) | 9477 |
| **Dynamic filtering** | **Small** | 37 (±6.5) | 67.1 (±8.92) | 1.84 (±0.45) | 3 |
| | **Medium** | 200.4 (±32.9) | 234,6 (±36) | 13.1 (±2.29) | 3 |
| | **Large** | 380.6 (±72) | 422 (±94.8) | 19.6 (±4.9) | 4 |
| **Semantic search** | **Small** | 23.9 (±4.12) | 64.8 (±11.2) | 3.28 (±2.44) | 3 |
| | **Medium** | 116.5 (±41.8) | 167.9 (±50.9) | 14.7 (±2.87) | 7 |
| | **Large** | 190.2 (±37) | 270.6 (±43.8) | 18.3 (±3.84) | 9 |

*6.2.2 Results.* Table 3 reports the execution times. As expected, the time increases according to the model size for the server computation and the query. The server time increases faster for the dynamic filtering (by 758% between small and large models) due to the propagation of the filter and the implementation choices. Concerning the query time, the mean time for the semantic zooming is higher than the others (701.1 ms for the large model). This is due to a higher number of elements the back-end sent to the front-end (9,477 elements). The semantic zooming is a passive interactive feature (*i.e.,* not explicitly triggered by the modeler) linked to the physical zooming, which is a function widely used by modelers. Such latency between user interactions and the application of the semantic zooming may negatively impact the usability of a modeler. The standard deviation of the semantic zooming query time for the large model is high, with a deviation of around 46.5%. This may be due to the network quality affected by the number of elements sent. Regarding the client, the different measured average times for a specific model size are closed. This is expected, since the three interactive features share the same algorithm to be applied over the concrete syntax. The exception is the semantic zooming that has higher values than others for the medium and large models. This is correlated to the number of elements returned by the back-end. Each of them may correspond to a concrete syntax element to be hidden, then an element impacting the client time.

The number of elements sent by the back-end for the dynamic filtering and the semantic search are much less than the semantic zooming. This is due to the nature of the different interactive features. The semantic zooming requires a description of the elements to be hidden. When the language designer wants to display only a small subset of the model, the number of elements to

be hidden can be large. Conversely, for the dynamic filter and the semantic search the language designer describes the elements they want to highlight. The objective of such interactive features is to enhance the readability or navigation of a model. Consequently, the number of elements may often be limited.

**RQ2 – Performance. Does the approach scale in terms of performance when the computation are only done on the back-end side?** The analyses were conducted on Sirius Web modeling workbenches, which perform computations on back-end only. This thus requires network queries from front-ends to back-ends. The empirical results show that latency may negatively impact the usability of such semantic-aware interactive features. This is especially true for interactive features sending a large amount of data such as the semantic zooming. Even if the latency remains lower for small and medium models, it is starting to become important for our large model with 47,297 elements. There is no doubt that this latency will become more important and more significant with larger models, such as those found in industry with several millions of elements. As an outcome, these results show that language workbenches should not entirely depend on back-ends to perform computations. This implies latency that reduces the usability of interactive features. This is not a limitation of our approach, since such interactive features require minimal latency, while some of them can be called up regularly or even passively. To consider interactive features that require quick results, language workbenches should also permit computations on the front-end side by, for example, loading a consequent part of the model under edition.

### 6.3 Threats to validity

**Scope.** The two use cases and the developed implementation are representative of the type of DSMLs our proposal supports, and can thus scope our proposal to discuss its generalization. Our approach targets language workbenches that: produce modeling environments for graphical DSMLs; follow the standard MDE approach (as detailed in Section 2); expose information related to its user interfaces and required to code an interactivity library. More precisely, concrete syntax of targeted DSMLs entity-relation diagrams.

**UI.** Regarding UI information, one may, for example, add a button in a toolbar, hence this toolbar must be identified. The language workbench can provide a dedicated UI model containing such information, as proposed in [67]. In this case the interactivity library can rely on this model. If the language workbench does not provide such an explicit UI model, the interactivity library must rely on UI information extracted from the language workbench code. This is the case of Sirius Web where we manually add the needed buttons in Sirius Web code.

**Generalization.** We developed our Sirius Web interactivity runtime based on the two case studies. The *Interactivity* DSL itself is independent of the language workbench, since the execution and the interaction with the UI of the modeling workbench itself are managed inside the interactivity library, which is specific to the language workbench. In future work, we will implement an interactivity library for other language workbenches such as Eclipse GMF.

**Scalability.** The evaluation of scalability depends on the implementation choice of our proposal and on the language workbench we used. This affected the empirical study we conducted to evaluate the scalability: Sirius Web is a cloud-native application where the front-end queries the back-end, where all the computation are done, to obtain model elements to display. So, the performance measures also include network latency and marshalling operations to transfer data between the

front-end and the back-end. We thus consider that our experiments with Sirius Web are the worst case scenario in terms of performance, since these features concerning the client must be fully computed on the server.

## 7 RELATED WORK

This section discusses related work and put them in relation with the challenges we identified in Section 3.2, namely: the semantic gap, the performance gap, the platform gap, and the UI gap.

### 7.1 Interactive features in language workbenches

Interactive features go beyond graphical DSMLs. For example, [70] surveys interactive features found in development environments of textual programming languages and DSMLs. Regarding textual DSMLs, LSP (Language Server Protocol) helps in decoupling front-ends and back-ends of Web modeling workbenches [51]. Research work proposed an extension of LSP to cover graphical modeling languages [22, 25, 61], we will call these approaches GLSP-like (Graphical LSP). LSP and GLSP-like approaches help developers in building modeling front-ends that externalize the execution of specific editing services (*e.g.,* auto-completion, code lens) to different back-ends. In particular, in [22] the authors proposed to supplement LSP with semantic zooming and off-screen visualization techniques. These research work mainly concern the *performance gap* and the *performance* RQ: GLSP-like work did not discuss their impact on performance and, therefore, on the usability of the produced modeling environment. However, our results show that semantic-aware features require computations that take enough time to possibly affect the usability of the modeling workbench. Such computation times can be increased by network queries when they are performed on a separated back-end. It can, for example, lead to a slow semantic zooming. This raises the need for language workbenches to better consider interactive features and the low latency they may require to be usable. For example, [39] proposed an approach for navigating within large models. This approach relies on fragmenting the models into several smaller ones. Such an approach may be interesting to optimize queries by reducing the size of a large model. Moreover, these work also concern the *semantic gap*: GLSP-like approaches do not help developers in the coding of interactive features, as discussed in [50]. Our approach aims at overcoming this issue.

### 7.2 Improving the usability of modeling workbenches

The final goal of our approach is to permit language designers to build modeling workbenches with interactive features designed to improve the usability. In this sub-section, we thus discuss approaches that also aim at improving the usability of modeling workbenches.

In [67], the authors proposed an approach that allows language designers to describe the user interface of a modeling workbench. This approach follows the same rationale as ours: language designers must be able to specialize the interactivity and the user interface of their modeling workbenches to improve their usability. Our approach complements the one proposed in [67]: we focus on the selection and specialization of interactive features to be used within the output modeling workbenches. Closely related, approaches proposed to explicitly specify the layout of graphical DSMLs [24, 32], and therefore, contribute to the specification of the user interface of modeling workbenches.

These two approaches concern the *UI gap*, as they propose new meta-languages dedicated to the description of UIs in the MDE development process of graphical DSMLs. Our approach would benefit from such approaches that explicitly expose a UI model.

CEViNEdit [37] is an approach that aims at easing the creation of graphical modeling workbenches. This approach improves related approaches with specific features to design cognitively effective visual notations. The workbenches that this approach produces embed mechanisms to

evaluate the graphical notation. Our approach complements this one with a focus on interactive features of graphical modeling workbenches. This approach focuses on another challenge we do not tackle in this paper: how to build a language workbench that monitor metrics to be used to evaluate its usability. This may help in evaluating the performance (see the performance RQ) of the integrated interactive features.

In [20], the authors detail the concept of blended modeling, which consists of editing a model through different concrete syntaxes. Closely related, model Sensemaking [48] is an approach that proposes specific visualization strategies for dedicated model understanding tasks. One goal is to provide users with better representations when performing specific tasks on large models. These two approaches focus on proposing different concrete syntaxes for DSML to ease the understanding of models. Our approach, rather, aims at augmenting graphical modeling workbenches with interactive features.

Various research work proposed editing services that aim at improving the productivity, and therefore the usability, of a modeling workbench. Those services include recommender systems [2–4, 42, 75], modeling assistance systems [33, 63, 68], auto-completion systems [47], and graphical model diffing [79]. Regarding the semantic change, these approaches propose services to be integrated as semantic-aware interactive features in modeling workbenches. These services rely on specific algorithms and techniques (*e.g.,* machine learning), therefore they cannot be part of our proposal. Instead, they complete our proposal by proposing semantic-aware interactive services.

Finally, research work investigated the user of virtual [78] or augmented [15] reality interfaces in modeling workbenches. The scope of our work currently focuses on classical graphical user interfaces.

### 7.3 Textual DSLs for specifying visualizations

Different research work proposed approaches to describe data visualizations using a grammar-like syntax: *GoTree* focuses on tree visualizations [44]; *Atom* on unit visualizations [55]; *Vega-lite* on data visualization with interactive features [62]. These approaches focus on selecting the type of visualizations (*e.g.,* sunburst, node-link) and the layout. Such approaches concern by the *semantic gap*. In particular, visualizations do not edit data, and do not consider the semantic of the tree nodes and links in the representation. Only *Vega-lite* [62] enables the customization of interactive features that occur when selecting graphical items. The other approaches come with predefined user interactions.

### 7.4 Model-driven UI development approaches

Our approach is based on the classical MDE development process of DSML. However, model-driven approaches were proposed to develop interactive systems. This sub-section aims at aligning the classical DSML development approach with the main ones related to interactive system development.

Cameleon is a framework to statically or dynamically adapt an interactive system from one context of use to another one [17]. Closely related, UsiXML [71] and Teresa [56] propose XML-based meta-languages to describe multi-platform interactive systems and multi-modal interactions. While our proposal is not related to dynamic adaptation, it is closely related to the separation between abstract and concrete user interfaces, as promoted by these approaches thereby concerning the *platform gap*. Our interactivity meta-language describes the interactive features of a given DSML, independently from the concrete platform (*e.g.,* Sirius Web).

Research work on model-driven UI also proposed architectures to separate user interactions from their resulting user commands [7, 9, 12]. Our approach relies on this principle where language

designers specify user commands using the proposed interactivity DSML, while the interactivity library maps those commands to user interactions.

## 8 CONCLUSION

This paper proposes a novel approach that helps language designers in building interactive features in their graphical modeling workbenches. Those features focus on specific modeling tasks, such as navigation tasks.

Our future work will deal with the integration of our proposal into GLSP approaches, in order to produce both GLSP servers and implementations of the interactive features.

We also aim at investigating other kinds of interactive features to improve the usability of modeling workbenches and we will work on improvements of our approach to give the capability to add new interactive features into dedicated libraries. In particular, we will work on helping language designers integrate modeling guidance [19, 74] in their modeling workbenches. Finally, we will implement our approach within other language workbenches.

## REFERENCES


[1] Silvia Abrahão, Francis Bordeleau, Betty Cheng, Sahar Kokaly, Richard Paige, Harald Stöerrle, and Jon Whittle. 2017. User experience for model-driven engineering: Challenges and future directions. In *Proc. of MODELS'17*. IEEE, 229–236.

[2] Lissette Almonte, Antonio Garmendia, Esther Guerra, and Juan de Lara. 2023. Reuse and Automated Integration of Recommenders for Modelling Languages. In *Proc. of SLE'23*. ACM, 97–110.

[3] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan De Lara. 2021. Recommender systems in model-driven engineering: A systematic mapping review. *Software and Systems Modeling* 21, 1 (2021), 1–32.

[4] Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. 2021. Automating the synthesis of recommender systems for modelling languages. In *Proc. of SLE'21*. ACM, 22–35. https://doi.org/10.1145/3486608.3486905

[5] Omar Badreddin, Rahad Khandoker, Andrew Forward, Omar Masmali, and Timothy C. Lethbridge. 2018. A Decade of Software Design and Modeling: A Survey to Uncover Trends of the Practice. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Copenhagen, Denmark) *(MODELS '18)*. Association for Computing Machinery, 245–255. https://doi.org/10.1145/3239372.3239389

[6] Eric Barboni, Rémi Bastide, Xavier Lacaze, David Navarre, and Philippe Palanque. 2003. Petri net centered versus user centered Petri nets tools. In *10th Workshop on Algorithms and Tools for. Petri Nets (AWPN 2003)*. Springer, 10 pages.

[7] Olivier Beaudoux, Mickaël Clavreul, Arnaud Blouin, Mengqiang Yang, Olivier Barais, and Jean-Marc Jézéquel. 2012. Specifying and Running Rich Graphical Components with Loa. In *EICS'12: Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*. 169–178. https://doi.org/10.1145/2305484.2305513

[8] Arnaud Blouin. 2024. A Type System for Flexible User Interactions Handling. *Proc. ACM Hum.-Comput. Interact.* 8, EICS, Article 246 (jun 2024), 27 pages. https://doi.org/10.1145/3660248

[9] Arnaud Blouin and Olivier Beaudoux. 2010. Improving modularity and usability of interactive systems with Malai. In *EICS'10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*. 115–124.

[10] Arnaud Blouin and Jean-Marc Jézéquel. 2022. Interacto: A Modern User Interaction Processing Model. *IEEE Transactions on Software Engineering* 48, 9 (2022), 1–20. https://doi.org/10.1109/TSE.2021.3083321

[11] Arnaud Blouin, Naouel Moha, Benoit Baudry, Houari Sahraoui, and Jean-Marc Jézéquel. 2015. Assessing the Use of Slicing-based Visualizing Techniques on the Understanding of Large Metamodels. *Information and Software Technology* 62, 0 (2015), 124 – 142. https://doi.org/10.1016/j.infsof.2015.02.007

[12] Arnaud Blouin, Brice Morin, Olivier Beaudoux, Grégory Nain, Patrick Albers, and Jean-Marc Jézéquel. 2011. Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation. In *ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. 85–94. https://doi.org/10.1145/1996461.1996500

[13] Dominik Bork and Giuliano De Carlo. 2023. An extended taxonomy of advanced information visualization and interaction in conceptual modeling. *Data & Knowledge Engineering* 147 (2023), 102209. https://doi.org/10.1016/j.datak.2023.102209

[14] Erwan Bousse, Dorian Leroy, Benoit Combemale, Manuel Wimmer, and Benoit Baudry. 2018. Omniscient debugging for executable DSLs. *Journal of Systems and Software* 137 (2018), 261–288.

[15] Léa Brunschwig, Rubén Campos-López, Esther Guerra, and Juan de Lara. 2021. Towards domain-specific modelling environments based on augmented reality. In *Proc. of ICSE-NIER'21*. IEEE, 56–60.

[16] Jordi Cabot, Robert Clarisó, Marco Brambilla, and Sébastien Gérard. 2018. Cognifying model-driven software engineering. In *Proc. of STAF'17*. Springer, 154–160. https://doi.org/10.1007/978-3-319-74730-9_13

[17] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Q. Limbourg, L. Bouillon, and Jean Vanderdonckt. 2003. A unifying reference framework for multi-target user interfaces. *Interacting With Computers* 15, 3 (2003), 289–308.

[18] Javier Cámara, Javier Troya, Lola Burgueño, and Antonio Vallecillo. 2023. On the assessment of generative AI in modeling tasks: an experience report with ChatGPT and UML. *Software and Systems Modeling* 22, 3 (Jun 2023), 781–793. https://doi.org/10.1007/s10270-023-01105-5

[19] Shalini Chakraborty and Grischa Liebel. 2024. Modelling guidance in software engineering: a systematic literature review. *Software and Systems Modeling* 23, 1 (2024), 249–265.

[20] Federico Ciccozzi, Matthias Tichy, Hans Vangheluwe, and Danny Weyns. 2019. Blended modelling-what, why and how. In *Proc. of Companion MODELS'19*. IEEE, 425–430.

[21] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. 2016. *Engineering modeling languages: Turning domain knowledge into tools.* CRC Press.

[22] Giuliano De Carlo, Philip Langer, and Dominik Bork. 2022. Advanced visualization and interaction in GLSP-based web modeling: realizing semantic zoom and off-screen elements. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems.* ACM, 221–231.

[23] Juri Di Rocco, Davide Di Ruscio, Claudio Di Sipio, Phuong T. Nguyen, and Alfonso Pierantonio. 2023. MemoRec: a recommender system for assisting modelers in specifying metamodels. *Software and Systems Modeling* 22, 1 (01 Feb 2023), 203–223. https://doi.org/10.1007/s10270-022-00994-2

[24] Aurélien Ducoin and Eugene Syriani. 2022. Graphical projectional editing in gentleman. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* ACM, 46–50.

[25] Eclipse. 2024. Graphical Language Server Platform. https://eclipse.dev/glsp/

[26] Sven Efftinge and Markus Völter. 2006. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit.* EclipseCon, 4 pages.

[27] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. 2008. Generating and evaluating choices for fixing inconsistencies in UML design models. In *Proc. of ASE'08.* IEEE, 99–108. https://doi.org/10.1109/ASE.2008.20

[28] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[29] Mathias Frisch and Raimund Dachselt. 2013. Visualizing offscreen elements of node-link diagrams. *Information Visualization* 12, 2 (2013), 133–162.

[30] Mathias Frisch, Raimund Dachselt, and Tobias Brückmann. 2008. Towards seamless semantic zooming techniques for UML diagrams. In *Proceedings of the 4th ACM Symposium on Software Visualization.* ACM, 207–208.

[31] Mathias Frisch, Jens Heydekorn, and Raimund Dachselt. 2010. Diagram editing on interactive displays using multi-touch and pen gestures. In *Diagrammatic Representation and Inference: 6th International Conference, Diagrams 2010, Portland, OR, USA, August 9-11, 2010. Proceedings.* Springer, 182–196.

[32] Hauke Fuhrmann and Reinhard von Hanxleden. 2010. Taming graphical modeling. In *Model Driven Engineering Languages and Systems: 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I 13.* Springer, 196–210.

[33] Miguel Gamboa and Eugene Syriani. 2019. Improving user productivity in modeling tools by explicitly modeling workflows. *Software & Systems Modeling* 18 (2019), 2441–2463.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley.

[35] Théo Giraudet, Mélanie Bats, Arnaud Blouin, Benoît Combemale, and Pierre-Charles David. 2024. Sirius Web: Insights in Language Workbenches - An Experience Report. *The Journal of Object Technology* 23, 1 (2024), 1–20. https://doi.org/10.5381/jot.2024.23.1.a6

[36] Michal Gordon and David Harel. 2010. Semantic navigation strategies for scenario-based programming. In *2010 IEEE Symposium on Visual Languages and Human-Centric Computing.* IEEE, 219–226.

[37] David Granada, Juan M Vara, Mercedes Merayo, and Esperanza Marcos. 2021. CEViNEdit: improving the process of creating cognitively effective graphical editors with GMF. *Software and Systems Modeling* 20 (2021), 867–895.

[38] John Hutchinson, Jon Whittle, and Mark Rouncefield. 2014. Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming* 89 (2014), 144–161.

[39] Antonio Jiménez-Pastor, Antonio Garmendia, and Juan de Lara. 2017. Scalable model exploration for model-driven engineering. *Journal of Systems and Software* 132 (2017), 204–225.

[40] Tomaž Kosar, Sudev Bohra, and Marjan Mernik. 2016. Domain-specific languages: A systematic mapping study. *Information and Software Technology* 71 (2016), 77–91.

[41] Oliver Köth and Mark Minas. 2002. Structure, abstraction, and direct manipulation in diagram editors. In *Diagrammatic Representation and Inference: Second International Conference, Diagrams 2002 Callaway Gardens, GA, USA, April 18–20, 2002 Proceedings 2*. Springer, 290–304.

[42] Tobias Kuschke, Patrick Mäder, and Patrick Rempel. 2013. Recommending auto-completions for software modeling activities. In *Proc. of MODELS'13*. Springer, 170–186.

[43] Krystle Lemon, Edward B Allen, Jeffrey C Carver, and Gary L Bradshaw. 2007. An empirical study of the effects of gestalt principles on diagram understandability. In *Proc. of ESEM'07*. IEEE, 156–165.

[44] Guozheng Li, Min Tian, Qinmei Xu, Michael J McGuffin, and Xiaoru Yuan. 2020. Gotree: A grammar of tree visualizations. In *Proc. of CHI'20*. ACM, 1–13. https://doi.org/10.1145/3313831.3376297

[45] Grischa Liebel, Nadja Marko, Matthias Tichy, Andrea Leitner, and Jörgen Hansson. 2018. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling* 17 (2018), 91–113. https://doi.org/10.1007/s10270-016-0523-3

[46] Hugo Lourenço, Carla Ferreira, João Costa Seco, and Joana Parreira. 2023. OSTRICH: a rich template language for low-code development (extended version). *Software and Systems Modeling* 22, 5 (Oct 2023), 1645–1663. https://doi.org/10.1007/s10270-022-01066-1

[47] Patrick Mäder, Tobias Kuschke, and Mario Janke. 2019. Reactive auto-completion of modeling activities. *IEEE Transactions on Software Engineering* 47, 7 (2019), 1431–1451. https://doi.org/10.1109/TSE.2019.2924886

[48] Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara. 2023. Model sensemaking strategies: Exploiting meta-model patterns to understand large models. In *Proc. of MODELS'23*. IEEE, 261–272.

[49] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[50] Haydar Metin and Dominik Bork. 2023. On Developing and Operating GLSP-based Web Modeling Tools: Lessons Learned from BIGUML. In *Proc. of MODELS'23*. IEEE, 129–139. https://doi.org/10.1109/MODELS58315.2023.00031

[51] Microsoft. 2024. Language Server Protocol. https://microsoft.github.io/language-server-protocol/

[52] Ángel Mora Segura, Juan de Lara, and Manuel Wimmer. 2024. Modelling assistants based on information reuse: a user evaluation for language engineering. *Software and Systems Modeling* 23, 1 (Feb 2024), 57–84. https://doi.org/10.1007/s10270-023-01094-5

[53] Gunter Mussbacher, Benoit Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, et al. 2020. Opportunities in intelligent modeling assistance. *Software and Systems Modeling* 19 (2020), 1045–1053.

[54] Mert Ozkaya and Deniz Akdur. 2021. What do practitioners expect from the meta-modeling tools? A survey. *Journal of Computer Languages* 63 (2021), 101030. https://doi.org/10.1016/j.cola.2021.101030

[55] Deokgun Park, Steven M Drucker, Roland Fernandez, and Niklas Elmqvist. 2017. Atom: A grammar for unit visualizations. *IEEE transactions on visualization and computer graphics* 24, 12 (2017), 3032–3043.

[56] Fabio Paternò, Carmen Santoro, Jani Mäntyjärvi, Giulio Mori, and Sandro Sansone. 2008. Authoring pervasive multimodal user interfaces. *Int. J. Web Engineering and Technology* 4, 2 (2008), 235–261.

[57] Parsa Pourali and Joanne M Atlee. 2018. An empirical investigation to understand the difficulties and challenges of software modellers when using modelling tools. In *Proc. of MODELS'18*. ACM, 224–234.

[58] Parsa Pourali and Joanne M Atlee. 2019. A focus+context approach to alleviate cognitive challenges of editing and debugging uml models. In *Proc. of MODELS'19*. IEEE, 183–193. https://doi.org/10.1109/MODELS.2019.000-3

[59] Tobias Reinhard, Silvio Meier, and Martin Glinz. 2007. An Improved Fisheye Zoom Algorithm for Visualizing and Editing Hierarchical Models. In *Second International Workshop on Requirements Engineering Visualization (REV 2007)*. IEEE, 10. https://doi.org/10.1109/REV.2007.2

[60] Tobias Reinhard, Silvio Meier, Reinhard Stoiber, Christina Cramer, and Martin Glinz. 2008. Tool support for the navigation in graphical models. In *Proceedings of the 30th International Conference on Software engineering*. ACM, 823–826.

[61] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. 2018. Towards a Language Server Protocol Infrastructure for Graphical Modeling. In *Proc. of MODELS'18*. ACM, 370–380. https://doi.org/10.1145/3239372.3239383

[62] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* 23, 1 (2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[63] Maxime Savary-Leblanc, Xavier Le Pallec, and Sébastien Gérard. 2023. Understanding the need for assistance in software modeling: interviews with experts. *Software and Systems Modeling* 23, 1 (2023), 1–33. https://doi.org/10.1007/s10270-023-01104-6

[64] Maxime Savary-Leblanc and Xavier Le Pallec. 2022. Interactive highlighting for digital UML class diagrams. In *SAM 2022 - System Analysis and Modelling (Co-located with MODELS 2022)*. ACM, 247–256. https://doi.org/10.1145/3550356.

3561557
[65] Christian Schenk, Sonja Schimmler, and Mark Minas. 2016. Operating diagram editors through unistroke gestures. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 21–25.
[66] Richard Sharp and Atanas Rountev. 2005. Interactive exploration of UML sequence diagrams. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 1–6.
[67] Vasco Sousa, Eugene Syriani, and Khady Fall. 2019. Operationalizing the integration of user interaction specifications in the synthesis of modeling editors. In *Proc. of SLE'19*. ACM, 42–54.
[68] Friedrich Steimann and Bastian Ulke. 2013. Generic model assist. In *Proc. of MODELS'13*. Springer, 18–34. https://doi.org/10.1007/978-3-642-41533-3_2
[69] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.
[70] Matúš Sulír, Michaela Bačíková, Sergej Chodarev, and Jaroslav Porubän. 2018. Visual augmentation of source code editors: A systematic mapping study. *Journal of Visual Languages & Computing* 49 (2018), 46–59.
[71] Jean Vanderdonckt. 2005. A MDA-Compliant Environment for Developing User Interfaces of Information Systems. *Lecture Notes in Computer Science* 3520/2005 (2005), 16–31.
[72] Gilles Vanwormhoudt, Mathieu Allon, Olivier Caron, and Bernard Carré. 2020. Template based model engineering in UML. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Virtual Event, Canada) *(MODELS '20)*. Association for Computing Machinery, 47–56. https://doi.org/10.1145/3365438.3410988
[73] Charlotte Verbruggen and Monique Snoeck. 2023. Practitioners' experiences with model-driven engineering: a meta-review. *Software and Systems Modeling* 22, 1 (Feb 2023), 111–129. https://doi.org/10.1007/s10270-022-01020-1
[74] Markel Vigo, Carmen Santoro, and Fabio Paternò. 2017. The usability of task modeling tools. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 95–99. https://doi.org/10.1109/VLHCC.2017.8103455
[75] Martin Weyssow, Houari Sahraoui, and Eugene Syriani. 2022. Recommending metamodel concepts during modeling activities with pre-trained language models. *Software and Systems Modeling* 21, 3 (2022), 1071–1089.
[76] Bianca Wiesmayr, Alois Zoitl, and Rick Rabiser. 2023. Assessing the usefulness of a visual programming IDE for large-scale automation software. *Software and Systems Modeling* 22, 5 (2023), 1619–1643.
[77] Insoo Woo, Sung Ye Kim, Ross Maciejewski, David S Ebert, Timothy D Ropp, and Krystal Thomas. 2009. SDViz: A Context-Preserving Interactive Visualization System for Technical Diagrams. *Computer Graphics Forum* 28, 3 (2009), 943–950.
[78] Enes Yigitbas, Simon Gorissen, Nils Weidmann, and Gregor Engels. 2023. Design and evaluation of a collaborative UML modeling environment in virtual reality. *Software and Systems Modeling* 22, 5 (2023), 1397–1425.
[79] Manouchehr Zadahmad, Eugene Syriani, Omar Alam, Esther Guerra, and Juan de Lara. 2022. DSMCompare: domain-specific model differencing for graphical domain-specific languages. *Software and Systems Modeling* 21, 5 (2022), 30 pages.