# Advanced Visualization and Interaction in GLSP-based Web Modeling: Realizing Semantic Zoom and Off-Screen Elements

Giuliano De Carlo
TU Wien, Business Informatics Group
Vienna, Austria
e1526998@student.tuwien.ac.at

Philip Langer
Eclipsesource
Vienna, Austria
planger@eclipsesource.com

Dominik Bork
TU Wien, Business Informatics Group
Vienna, Austria
dominik.bork@tuwien.ac.at

## ABSTRACT

Conceptual modeling is widely adopted in industrial practices, e.g., process, software, and systems modeling. Providing adequate and usable modeling tools is essential for the efficient adoption of modeling. Metamodeling platforms provide a rich set of functionalities and maturely realize state-of-the-art modeling tools. However, despite their maturity and stability, most of these platforms only slowly – if at all – leverage the full extent of functionalities and the ease of exploitation and integration enabled by web technologies. With the Graphical Language Server Protocol (GLSP), it is now possible to realize much richer, advanced opportunities for visualizing and interacting with conceptual models. This paper presents a concept and a prototypical implementation of two advanced model visualization and interaction functionalities with the Eclipse GLSP platform: *Semantic Zoom* and *Off-Screen Elements*. We believe such advanced functionalities pave the way for a prosperous modeling future and spark innovation in modeling tool development.

## CCS CONCEPTS

• **Software and its engineering** → **System description languages**; • **Human-centered computing** → **Interactive systems and tools**.

## KEYWORDS

Modeling tools, Web Modeling, Language Server Protocol, Visualization

**ACM Reference Format:**
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 *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550355.3552412

## 1 INTRODUCTION

Technology usage forms an essential part of our private and professional lives. Accessing the right tools and knowing how to use them correctly can save time and effort. The connection between the user of a tool and the tool itself is usually its user interface and the supported interactions that come with it. While the functionalities of a tool also play a significant role, without a graphical user interface that offers good visualization of its information, tools are often labeled as not very useful [17, 31]. This is especially important in conceptual modeling, where information visualization makes up a central aspect that directly influences the comprehensiveness of models and the usability and ease of use of modeling tools [11, 35]. Tool development is therefore denoted as an essential part of enterprise and business information systems modeling [11, 35] research. However, past research primarily focused on the development of new and the evaluation [4, 34] and improvement of existing modeling languages [5].

Today, model engineering has many different tools at its disposal. Most of these tools are mature applications that have been actively worked on over a relatively long period but have barely evolved in recent years [16, 17]. Their functionalities are often built on older technology stacks, i.e., they are not compatible with state-of-the-art web technologies. Adding advanced visualization and interaction functionality to them is often impossible without changing the software's underlying foundation, resulting in considerable development overhead. Although the results produced with such tools are still unsurpassed, the functionality, especially concerning information visualization and interaction, often lacks advanced techniques. A recent study yielded one of the fundamental challenges of modelers engaged with model-driven engineering is *"remembering contextual information"* [31, p. 233].

Advanced techniques like semantic zooming and off-screen elements could speed up the model development process. It could also improve usability and ease of use of the tools and comprehension of conceptual models by humans. Web technologies have been heavily used and improved over the previous years and offer a wide range of great functionalities, which is why they are the perfect fit to develop such advanced techniques. Compared to platforms used in most traditional modeling tools, web technologies provide a future-proof, feature-rich, robust, and efficient foundation for state-of-the-art visualization and interaction techniques. This work takes the first steps toward realizing *semantic zoom* and *off-screen elements* as concrete advanced visualization and interaction techniques with the Eclipse Graphical Language Server Protocol (GLSP) platform[1], thereby contributing concepts and prototypical solutions to the identified research challenges [31].

*MODELS '22, October 23–28, 2022, Montreal, QC, Canada*
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9466-6/22/10.
https://doi.org/10.1145/3550355.3552412

[1]The resulting artefacts are open source. The latest version via: https://github.com/glsp-extensions, this paper's version permanently via: https://doi.org/10.5281/zenodo.7007921

In the remainder of this paper, Section 2 briefly introduces the foundations and reports on related works. The concepts for realizing semantic zoom and off-screen elements are introduced in Section 3. The implementation of these concepts is reported in Section 4 and evaluated in Section 5. Eventually, Section 6 concludes the paper and provides directions for future research.

## 2 FOUNDATIONS AND RELATED WORK

In this section, we briefly introduce the foundations of the Eclipse Graphical Language Server Platform (Eclipse GLSP) and advanced model visualization & interaction before reporting on related works.

### 2.1 Eclipse GLSP

Even though the traditional client-server architecture started to gain acceptance in the late 1980's [36] and has since been applied to many significant developments such as the web, only recently has it reached the world of software development in the form of language servers. The language server protocol (LSP), recently introduced by Microsoft, RedHat, and Codeenvy in 2016 [6], is gaining interest in the scientific community. LSP splits today's heavy-weight monolithic IDE approaches into a client and a server. LSP standardizes the communication between these two components to synchronize client and server. Currently, version 3.17 of the protocol describes 40 different messages between client and server and has an implementation for over 100 different programming languages/technologies [25, 26].

Initially, LSP has only been defined and used for text-based languages. Still, it was quickly discovered that this concept could also be applied to other areas, one of them being graphical languages [33]. The Eclipse Graphical Language Server Platform (Eclipse GLSP) [10] is an open-source framework that uses an LSP-like protocol to enable diagram editing via a client/server architecture. The server is responsible for model management, the model logic, validation, and applying changes to the model(s). It supports the Eclipse Modeling Framework (EMF), based on which many modeling languages and their language-specific logic are already implemented. The server is written in the programming languages Java or TypeScript, exposing an inter-process communication interface via WebSocket or TCP socket connection.

The client is mainly responsible for rendering the graphical representation of a model and handling user interactions. Graphical elements are rendered as an SVG element inside a browser with the help of Eclipse Sprotty[2]. User interactions with the client may result in actions, which are, depending on their type, either handled locally on the client, e.g., for panning, zooming, or visual feedback, or they are transferred to the server, e.g., to perform a manipulation of the underlying model(s). If an action on the server results in a model change that affects the diagram, the server processes the change. It sends a new version of the diagram to be rendered with an *UpdateModelAction* back to the client to refresh the diagram view.

The client/server communication utilizes a protocol similar to Microsoft's LSP. As mentioned above, the communication between both entities is based on actions (see Figure 1). While many actions are reused from Sprotty, e.g., for model transfer and client-local

[2]https://github.com/eclipse/sprotty (Accessed: 16.05.2022)

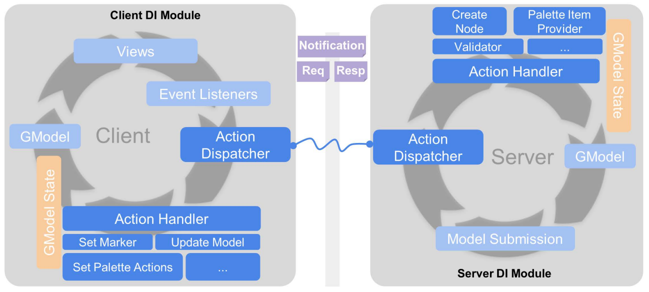

**Figure 1: Eclipse Action Life-cycle, adapted from[4]**

actions, many new ones are also added. They include model-specific actions, such as *CreateNodeOperation* which adds a node to the current model, editor-specific actions, such as *SetClipboardDataAction*, which copies data to the clipboard, and *Undo-/RedoOperation*. A full list of operations can be found in the GLSP protocol specification[3].

### 2.2 Advanced Model Visualization & Interaction

Under *advanced visualization and interaction features*, we understand original means that go beyond the traditional methods seen in almost all user interfaces of today, e.g., dynamic, context-sensitive, and interactive representation. While current modeling tools mostly lack such features (cf. [7] for a recent survey), as users, we are used to working with such tools (e.g., zooming in Google Maps or computer games [20]). The Level of Detail (LoD) forms an integral part of such advanced functionality and will be the main topic in this work. LoD enables the differentiated representation of model elements depending on various contextual factors such as, e.g., the current zoom level, the distance (to other elements), or the importance of an element. The motivation behind these approaches is that they can help reduce the complexity of the displayed information in certain situations and foster efficient human processing.

Using multiple LoDs goes back to works like Donelson's [8] which describes an information management system with multiple displays, one showing specific information and the other showing a shrunken version to help with navigation. Another influential work is the interface model "Pad" [30] from 1993. Pad is an interface shared among users, showing information in multiple LoD. It uses different views- or portals- to show more details about specific parts of an information source. Based on these concepts, many other ideas were proposed, among them are fisheye views [1, 14, 32], semantic zooming [13, 23, 24, 29, 30], lenses [38], and off-screen elements [40].

### 2.3 Related Work

In this section, we report on findings related to our notion of advanced visualization and interaction. We are thus interested in methods that, unlike, e.g., basic zooming or scrolling, were developed more recently and ideally added a noticeable benefit to the usage of a tool. Because of the wide range of tools that exist nowadays, many exciting and advanced features can be found in all kinds

[3]https://github.com/eclipse-glsp/glsp/blob/master/PROTOCOL.md (Accessed: 06.05.2022)
[4]https://www.eclipse.org/glsp/documentation/actionhandler/ (Accessed: 06.05.2022)

of domains. To name an example outside of the model engineering field: Many code/text editors expanded the scroll bar and, instead of simply showing the typical blanc bar, displayed a miniature version of the file that is currently opened to increase spatial awareness of the workspace. Others use the scroll bar to show indicators for, e.g., contextual information about off-screen elements or search results.

Narrowing the scope to diagramming tools, we have tools like yEd[5], MS Visio[6], or Visual Paradigm[7], which offer features such as creating groups of diagram elements, expanding/collapsing model elements, model decomposition, and first realizations of zooming.

Not many of such advanced features are provided by (meta-)modeling platforms yet [7]. While some of them do offer advanced features, such as model decomposition in MetaEdit+[8] or Enterprise Architect[9], or a minimap of the current workspace in the Eclipse Modeling Tools[10] or MetaModelAgent[11], they do not provide any recent state-of-the-art functionalities like semantic zooming.

Because of this gap, we looked at recent literature to find potential features that can be added to (meta-)modeling platforms. Our findings included various exciting features in different categories, such as speed-dependent automatic zooming [21] or onion-graphs [22]. We considered the most promising categories: *semantic zooming* and *off-screen elements*. Visualizing off-screen elements improves spatial awareness and may even provide quick means of navigating the workspace. Different variations of this concept can be found throughout the literature [2, 12, 18, 19, 38, 40]. Semantic zooming provides the ability to keep the workspace clean by only showing the most relevant information for the current zoom level (i.e., LoD). It has been applied to tools in different fields, among them software development [37, 39], parallel computing [23], video editing [24] or text documents [9, 29], and UML diagrams [13]. We believe semantic zoom and off-screen elements are good first candidates to mitigate the *"remembering contextual information"* problem [31, p. 233].

## 3 CONCEPT

This section introduces the concepts we developed to realize semantic zoom and off-screen elements on a GLSP-based environment.

### 3.1 Semantic Zoom

Semantic zooming allows a user to change the graphical representation of a model by zooming in or out. Depending on the current zoom level, an object can either show more or fewer details but always remains visible in one form or another. At the lowest zoom level, this could, e.g., only be their title or outlines. The further the user zooms in, the more details are seamlessly added until all details are shown at the highest level. An example can be seen in Figure 3, which shows the same model element in four different levels of detail.

[5]https://www.yworks.com/products/yed (Accessed 06.05.2022)
[6]https://www.microsoft.com/en-us/microsoft-365/visio (Accessed: 06.05.2022)
[7]https://www.visual-paradigm.com/ (Accessed: 06.05.2022)
[8]https://www.metacase.com/mep/ (Accessed 06.05.2022)
[9]https://sparxsystems.com/products/ea/index.html (Accessed: 06.05.2022)
[10]https://www.eclipse.org/downloads/packages/release/2022-03/r/eclipse-modeling-tools (Accessed: 06.05.2022)
[11]https://www.metamodelagent.com/ (Accessed: 06.05.2022)

*3.1.1 Level of Detail.* The idea behind this concept is to allow language server developers to define an arbitrary number of discrete LoDs on the server. These levels can then be used throughout the server and the client. They consist of a name and a zoom level range (*from* and *to*) defined as $[from, to[$. This range is used to determine when a discrete LoD is active. If *from* is omitted, it is treated as $-\infty$, if *to* is omitted, it is treated as $\infty$. Developers can use their interpretation of a zoom level and these values. However, because the zoom level is exchanged between client and server, both must have the same interpretation. In the implementation of this prototype, the zoom level is a number $x > 0$ where 1 is considered the default zoom level. Everything above 1 is zoomed out, and everything below 1 is zoomed in. This is done because Sprotty uses the natural exponential function on the *deltaY* value of the standardized UI DOM events to calculate the current zoom level of the viewport. The four defined discrete LoD (as seen in Figure 3 left to right) are as follows: *overview* $[1.25, \infty]$, *intermediate* $[0.5, 1.25)$, *intermediate detail* $[0.25, 0.5)$, and *detail* $[0, 0.25)$. Developers are responsible for the correct and complete range coverage; gaps or overlapping levels may lead to undefined behavior in the current implementation.

All defined discrete LoDs can be requested by the client with the *RequestDiscreteLevelOfDetail* action. Because they are needed to render the model, this action is usually one of the first dispatched to the server. The server responds with a *SetDiscreteLevelOfDetail* action, which includes all discrete LODs in the JSON format. This information only has to be requested once and can be cached by the client because it is not subject to change.

*3.1.2 Rules.* LoD rules are used to trigger specific behavior on certain LoD levels. These rules describe how certain graphical representations should be adjusted when the client enters a specific LoD. All rules consist of at least a type and information about when it is supposed to be applied. Furthermore, depending on the type of the rule, they can include additional rule-specific parameters.

*Rule types.* Currently, three types of rules exist which define the rule behavior and which are transferred in the *type* field:

- CssStyleRule: This rule is mighty and can accomplish most of the graphical adjustments alone by applying certain CSS styles to objects (e.g., to increase the font size of text when a user zooms out, change the background color, or add transparency to elements). An example can be seen when comparing Figure 3a to 3b, which shows the same information about an object but with dynamically adjusted font size. Additionally, the value of a given CSS-style can include the keyword *'$clevel'*. On the client, all occurrences of this keyword are automatically replaced with the current zoom level. This allows making values dynamically dependent on the current continuous zoom level, e.g., to increase the font size with every zoom-in event.
- VisibilityRule: It allows hiding specific objects with the additional boolean parameter *setVisibility*. This rule is important and used often to completely remove/add specific elements (e.g., properties) of an object when the user zooms out/in (cf. Figure 3b and 3d).

- LayoutRule: It allows modifying an element's layout (e.g., padding, horizontal/vertical gap, or a minimal width/height), which is usually defined by the server. It can, for example, be used to increase the padding of certain elements at zoom levels that offer a lot of space (cf. Figure 3a to 3b).

*Rule trigger.* The information about when a rule is supposed to be applied is stored inside the *trigger* field. It can hold either a *triggerDiscreteLevel*, or a *triggerContinuousLevel*, depending on whether it should be triggered on a discrete LoD, or a continuous zoom level. Discrete LoDs are references to the LoDs which were transferred initially with the *SetDiscreteLevelOfDetail* action, and continuous levels are two double values that specify a range in which the rule is supposed to be applied. Most of the time, rules reference a discrete LoD. This keeps all rules grouped under the label of a discrete LoD, making it easy to change the range in which all rules are applied by simply adjusting it once in the discrete LoD.

*Rule application.* Additionally to the information about the type of a rule and its trigger, a reference to elements that it is applied to has to be supplied. This can be done on the server by instantiating rules and assigning them to elements by their GModel element type. Two types of rules exist those that are applied to the model at the client and those that are applied to the model at the server.

*Client rules.* Most rules are executed on the client because the client has the information about the current zoom level, and it is the client's responsibility to render objects accordingly. Nevertheless, the assignment of rules is language-specific information, which is why they are defined and stored on the server. Since the client only has information about how to apply specific rules but not their assignments, these assignments must be transferred to the client, along with the model itself. This is done with a new *RequestLevelOfDetailRules* action to which the server responds with a *SetLevelOfDetailRules* action. Similarly to requesting the discrete LoDs, the client requests all LoD rules and assignments once and caches the response.

Client rules are applied during the rendering process on the client. The logic of a client rule is part of the client itself; the server merely provides information about what rule to apply and when. Language server developers can change this logic at will, and new rules can easily be added to the client. The client also has access to the current continuous zoom level during the rendering process. This means that a rule can integrate the current zoom level into its logic and continuously adjust certain elements.

*Server rules.* Server rules are used in cases where a rule has to be applied on the GModel before it is sent to the client. Usually, these are fundamental changes to the layout that the client cannot make. The disadvantages of server rules are that the server does not have information about the current zoom level of the client. By default, the server assumes a zoom level of 1. Additionally, the client can provide the current zoom level as an optional field in the *requestModel* action, which makes the server apply all rules for that zoom level in the resulting *setModel* action. Not only does the client have to provide the current zoom level, but it also has to notify the server whenever a server rule is supposed to be applied. For this reason, all server rules are also transferred to the client, along with all client rules. Whenever the client encounters a server rule, it has to request the model again to make the server apply the server rule. Because of this additional server round trip, server rules should be avoided when possible.

An example where a server rule would be required is the adjustment of the layout option *resizeContainer.* It tells the client whether to resize the parent of an element if the element becomes too large. If this were switched from *false* to *true* via a client rule, the client would not have correct bounds for this element because these are calculated on the server.

## 3.2 Off-Screen Elements

The off-screen elements feature allows users to see elements, even when not positioned inside the current viewport. This is especially useful when working with large models or models that show many details about their elements, as users tend to zoom in further. Zooming narrows the view that a user has on a model, which effectively pushes elements off-screen. As soon as an element becomes completely invisible they are replaced by smaller indicator elements which are pinned to the border of the viewport to keep contextual information (cf. [12]). This increases the sense of orientation of users, even with large models or while zoomed in.

*Nodes.* All nodes that are moved off-screen are replaced by smaller indicators pinned to the viewport's border. As soon as the original elements become visible, their indicator is replaced by the original element. Each GModel element type can have its indicator. This can be used to encode additional information into the indicators by, e.g., changing their form or color accordingly. A visual example of three off-screen indicators is given in Figure 5b. The color of each indicator is used to encode the information about whether it is an automated or manual task.

Overlapping indicators at the same place of the border are merged to prevent cluttering. The visual representation of merged indicators differs from the others (see Figure 5d). They use a white background color and include the number of merged indicators.

*Edges.* Nodes in diagrams are identified not only by their name but also by their position and relationships. For this reason, edges between elements play an important role in combination with off-screen element visualization. They are vital to identifying certain elements and keeping the mental map of a workspace intact. Although edges are also considered elements of a model, they do not possess off-screen indicators. When an element disappears from the viewport, its indicator element serves as the new port for all incoming and outgoing edges. In combination with their position, this helps the user quickly identify an off-screen element, even without visualizing their name. An example is given in Figure 5c, which shows two edges connected to off-screen indicators.

*Proxies.* All indicators also act as proxies for the elements they represent. All actions that can be performed on the original elements should also be able to be performed on their proxies. For example, connecting an edge from an on-screen element to a proxy of an off-screen element should create an edge from the on-screen element to the off-screen element represented by the proxy. In many cases, this decreases the number of actions that must be performed to achieve a specific goal, i.e., a user won't need to zoom out first to make both elements visible to connect them.

*Navigation.* All indicators also help navigate the model. Clicking on an indicator will automatically move the viewport to center the represented element, and select it. This lets the user immediately identify the element after zooming/panning and start working with it. When the user clicks on a merged indicator, all elements are selected. The client automatically calculates and sets the zoom level of the viewport to a value in which all selected elements are visible. This makes traversing a model easy and fast because the user does not have to zoom and pan to the viewport manually.

## 4 PROTOTYPICAL IMPLEMENTATION

In the following, we report on our efforts to realize prototypical implementations of the two concepts for semantic zoom and off-screen elements we introduced in Section 3 using the Eclipse GLSP platform. Consequently, this section also serves as a technical feasibility evaluation of the proposed concepts using one concrete LSP-based tool development environment.

### 4.1 Semantic Zoom

The semantic zoom feature required adjustments on the client, the server, and additions of new actions to the LSP protocol. Figure 2 shows a sequence diagram of the most critical operations to realize Semantic Zoom in GLSP modeling tools. It shows the client's initialization process, which requests all required information from the server, followed by operations performed during the rendering process of a model.

*4.1.1 Server.* The server is built exclusively in Java. The following section will give an overview of the prototype's architecture and its functionalities.

*Discrete Levels of Detail.* All discrete LoDs are defined inside an enumeration file on the server. Each enumeration entry consists of an LoD name and two double values *from* and *to*, which define the zoom level interval in which the rule is active.

*Actions.* The semantic zoom feature required two new actions that are handled by their respective handler class on the server: *RequestDiscreteLevelOfDetail*, and *RequestLevelOfDetailRules*.

- *RequestDiscreteLevelOfDetailActionHandler*: It fetches all discrete LoDs that were defined and converts them into a JSON object. This object is then sent to the client in an *SetDiscreteLevelOfDetail* action.
- *RequestLevelOfDetailRulesActionHandler*: It fetches all registered rules, along with the ids or selectors of the model elements assigned, and converts them into a JSON object. This object is then sent to the client in an *SetLevelOfDetailRules* action.

*Rule Registry.* An example of a rule registration can be seen in Listing 1 which changes the bottom and top border size. Rules are registered for certain model element types via a selector, similarly to CSS selectors, e.g., for all automated and manual tasks in the example below. Furthermore, a trigger is added to the rule, which activates it when the client enters the discrete level of detail "Overview". All rules registered in the rule registry are transferred from the server to the client, which then registers the triggers according to the rule specifications.

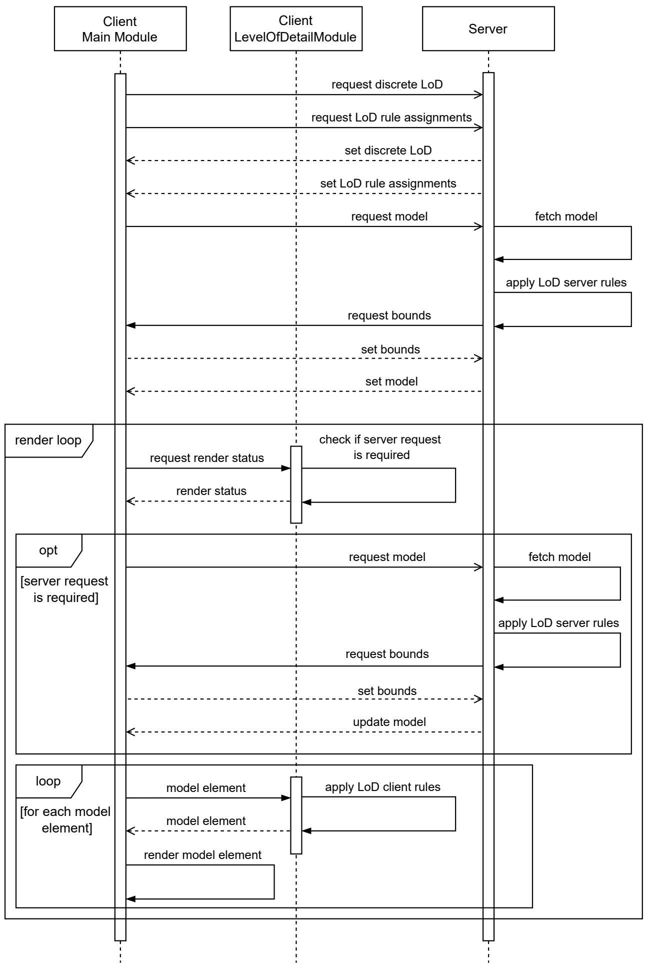

**Figure 2: Essential GLSP operations for Semantic Zoom.**

```
LayoutRule rule = new LayoutRule(new GLayoutOptions()
    .paddingBottom(3D)
    .paddingTop(3D)
);

rule.addLevelOfDetailRuleTrigger(
  new LevelOfDetailRuleTriggerDiscrete()
    .addDiscreteLevelOfDetail(
      DiscreteLevelOfDetailEnum.OVERVIEW
    )
);

registerRule(
  ModelTypes.AUTOMATED_TASK + "," +
       ModelTypes.MANUAL_TASK,
  rule
);
```

**Listing 1: Java code that shows the registration of a new rule.**

*Rules.* As already explained in Section 3.1, three concrete rules currently exist: *CssStyleRule*, *VisiblityRule*, and *LayoutRule*. Depending on whether a rule is a server- or client rule, they inherit from different interfaces/abstract classes. On the server, client rules only require a simple structure because their logic is implemented and applied to the client. The respective action handler understands their structure, and, once registered in the registry, they are sent to the client if requested. On the other hand, server rules are handled slightly differently. Not only does the server require additional logic to apply them, but they also have to be transferred to the client in a different form than the client rules. Unlike client rules, all the client needs to know about a server rule is that it is a server rule and when to activate it. No additional rule-specific parameters are needed because the client does not have to execute any rule-specific logic. This is realized with multiple different interfaces for server rules. Each server rule requires a rule-specific implementation of its logic on the server. Before it is transferred to the client, it is stripped from this logic and all other information that is only relevant to the server.

The server applies server rules inside the *ModelSubmissionHandler*. This handler is responsible for sending actions to the client that update or set the current model. Before such a message is sent to the client, all server rules are applied to the model. This is done by traversing the entire GModel tree and checking each element for referenced rules. If rules exist for an element, it is checked whether the rule is currently triggered. In case it is triggered, the rule-specific logic is applied to the model element.

Information about when a rule is supposed to be applied is stored in separate classes. Both ways of triggering a rule, by discrete and continuous zoom levels, are represented by different concrete classes that inherit from the same abstract class. The main difference between them is that the continuous rule trigger requires two parameters *from* and *to*, while the discrete rule trigger only requires a reference to a discrete LoD.

#### 4.1.2 *Client.*

The architecture of the client is very similar to that of the server. Most classes and interfaces defined on the server can also be found on the client.

*Discrete Levels of Detail.* Discrete levels of detail are fetched via an action and then stored on the client for the entire remaining session. Each discrete LoD is stored inside an instantiated *DiscreteLevelOfDetail* class and consists of, similarly to the server, three variables: *from*, *to*, and *name*.

*Actions.* Two new actions were added that are handled by the client: *SetDiscreteLevelOfDetail* and *SetLevelOfDetailRules*. Both actions are handled by their respective handler classes:

- *SetDiscreteLevelOfDetailActionHandler*: It takes the received JSON object and converts all discrete LoDs into *DiscreteLevelOfDetail* TypeScript class objects. These objects are then stored and used throughout the remaining session. Furthermore, when this action is received, the current discrete level of detail is determined and stored by fetching the current continuous level of detail of the stage and converting it into a discrete one.
- *SetLevelOfDetailRulesActionHandler*: It takes the received JSON list of all rules defined on the server and converts them into their respective TypeScript classes. Like the discrete LoDs, they are stored and accessed throughout the remaining session.

*Rules.* For the client to understand all rules that were received during the *SetLevelOfDetailRules* action, they have to have an implementation on the client. All rules inherit from a set of interfaces/abstract classes. These interfaces/classes expose the basic fields required in all rules. Namely *type*, which holds the unique type of the rule; *isServerRule*, which is used to tell whether a rule is a server rule; and *trigger*, which holds information about when this rule is supposed to be applied. The information in the field *trigger* is stored in one of two concrete class implementations inherited from the same abstract class. It can either be triggered on a continuous or discrete zoom level. All in all, the concept and implementation of trigger information are very similar to that of the server. Furthermore, each rule also has a set of common functions, for example, a *handle()* function, which holds the logic that applies a rule to the model elements. It takes a graphical element about to be rendered on stage as an argument. Rule-specific logic can then be applied to this element inside the function before it is returned and rendered. Currently, the following logic is applied to existing rules:

- *CssStyleRule*: It appends all CSS styles of the rule to the received graphical element.
- *VisibilityRule*: Depending on whether the rule-specific parameter *setVisibility* is *true* or *false*, this rule adds or removes the CSS class *hidden* to the graphical element. This CSS class sets the CSS *display* property to *none*, which removes the element from the stage.
- *LayoutRule*: It appends all layout options of a rule to the received graphical element.

Each type of rule or rule trigger must be registered for the client to understand and use. Unlike the rule registry on the server, registering a rule on the client only makes the client aware that a rule exists by creating a connection between a unique type and its concrete class. It does not create connections between rules and model elements. This is done only on the server and then fetched by the client. Once a rule or rule trigger is registered on the client, the client can initialize instances of it when necessary.

All rules and their assignments are stored in the central *LevelOfDetail* class, which can get injected into and used by many other classes throughout the client. It is a singleton instance that is instantiated during the client's initialization. Among its responsibilities are converting rule JSON objects to TypeScript objects, conversion of continuous levels of detail to discrete levels, storing rules and their assignments, and looking up and returning all assigned rules for an object id which involves the evaluation of selectors.

*Zoom Listener.* The zoom listener is another new addition to this prototype. It inherits from the already existing *MouseListener* and is an event-listener that listens to the standard "wheel" DOM event. Every time a *wheel* event is triggered, the listener fetches the current zoom level of the viewport. In the current implementation, the zoom level is calculated with the event's *deltaY* value, representing the vertical scroll amount of the performed event. The zoom listener keeps a copy of the last zoom level in memory. This copy always represents the current zoom level of the viewport and can be fetched by other modules in case they need it. Currently, only the mouse wheel can be used to increase or decrease the zoom level.

*Rendering.* Once the client has all information about LoDs, rules, and their assignments, it can start the rendering process in the class *LevelOfDetailModelRenderer*, which inherits from the default class *ModelRenderer*. Furthermore, the class *LevelOfDetailRenderer* is used to apply rules.

- *ModelRenderer*: It is the default implementation that is used to render model elements. Among others, it consists of the two important functions *renderChildren()* and *renderElement()*. The *renderChildren()* function calls *renderElement()* on all children of an element. The *renderElement()* function then renders the individual view-element.
- *LevelOfDetailModelRenderer*: It overwrites the default *renderElement()* function and adds additional functionality. In the function, when the root element is about to be rendered, it is checked if a new server round trip has to be made. This can occur on two occasions: ($i$) the client encountered a server rule which has to be applied, and ($ii$) the client encountered a rule which requires the server to re-create the model layout. ($ii$) occurs when the server is responsible for the sizes of objects, and the client tries to apply a rule which changes this size. In both cases, the client sends a new *RequestModel* action to the server. The server applies all server rules, recalculates the general sizes and positions, and sends the model back to the client. Currently, a switch between discrete levels of detail always triggers a new server round trip to recalculate sizes. Another new functionality added by the *LevelOfDetailModelRenderer* is a call to the class *LevelOfDetailRenderer*, which applies all relevant rules to model elements.
- *LevelOfDetailRenderer*: This renderer consists of two functions *checkForRerender()*, and *prepareNode()*. The function *checkForRerender()* is used to check if a new server round trip is required. It does this by checking if any rule has become active in the last zoom event, and no server round trip has been made because of it yet. The *prepareNode()* function is used to apply all relevant rules on a model element by calling the specific rule's implementation of its *handle()* function.

Figure 3 shows several screenshots of the Semantic Zoom prototype. Each screenshot shows the LoD-specific rendering of the model content from a very high level representation (Figure 3a) until a very detailed representation (Figure 3d). The representation is automatically adjusted based on the current LoD.

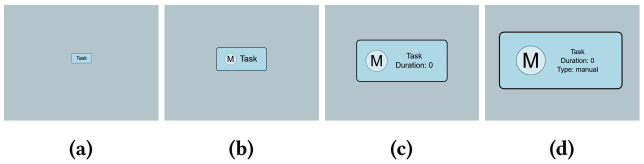

**Figure 3: Screenshots of the Semantic Zoom prototype, showing the model at four different zoom levels.**[12]

## 4.2 Off-Screen Elements

The realization of the Off-Screen elements feature only consists of client-side functionalities. Except for small additions in HTML and CSS, the prototype is built entirely in TypeScript. The following section will overview the most relevant algorithms, interfaces, and implementation classes. Figure 4 shows a sequence diagram of the most critical operations we used to realize Off-Screen Elements with GLSP.

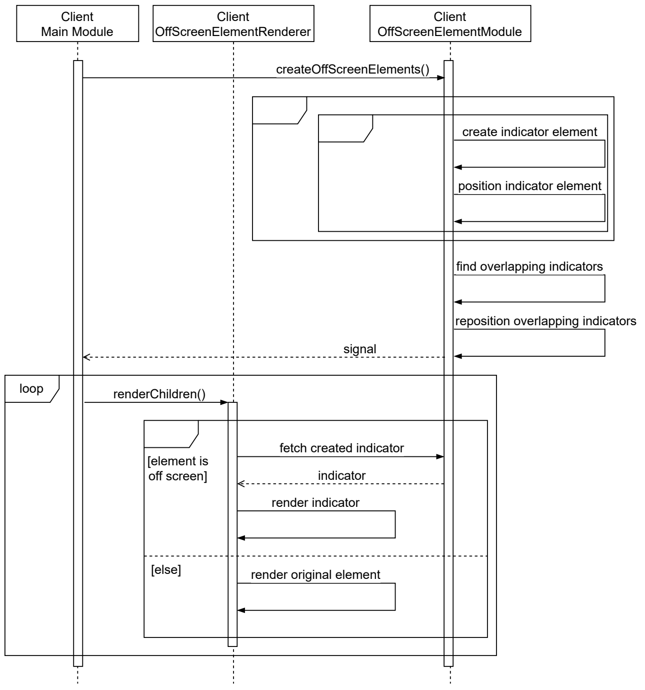
Client Main Module
Client OffScreenElementRenderer
Client OffScreenElementModule
createOffScreenElements()
create indicator element
position indicator element
find overlapping indicators
reposition overlapping indicators
signal
loop
renderChildren()
[element is off screen]
fetch created indicator
indicator
render indicator
[else]
render original element

**Figure 4: Essential GLSP operations for Off-Screen Elements.**

*Indicators.* Each original model element, which is supposed to have off-screen indicators, must have a defined model and view. The model acts as the SModel element, which will be displayed, and the view is used to render the SModel element. Each model must be registered for the client to be aware of it. This registration connects the original SModel type, the SModel indicator element, and the indicator view definition. Each registration tells the client to replace all elements of that type with the specified off-screen indicator once they are moved off-screen.

*Positioning Indicators.* Indicators are always positioned at the border of the viewport closest to the element it represents. We developed an algorithm that calculates the position by checking which side of the viewport the invisible element is (i.e., left/right, top/bottom). If the element is at the left or the right, the y-coordinate of the indicator is set to the same value as the y-coordinate of the element. Otherwise, the x-coordinate is set – the respective other coordinate remains unchanged. Furthermore, they will be restricted from going below or above the bounds of the viewport to prevent the indicator from disappearing off-screen.

*Overlapping Indicators.* We developed an algorithm that identifies and combines a group of overlapping indicators. If a group is identified, the average position of all represented elements is calculated, which will be the position of the new merged indicator element. During manual tests of the overlapping implementation, we realized that, while moving the viewport around, elements were

[12]Semantic Zoom prototype video: https://www.youtube.com/watch?v=iBs-fGwq15Y

popping in and out because the indicator elements' calculated position is constantly adjusted during panning events. In situations where many indicators are close together, this adjustment causes them to merge and separate often. To mitigate this behavior, a configuration variable *OVERLAP_SIZE_MULTIPLIER* was added, which can be used to increase or decrease the size of an indicator element only during the calculation of existing overlaps. For example, a value of 2 would cause all indicators to appear twice as large to the algorithm during the calculation, which causes indicators to form larger groups and be merged more quickly. A value of 4 seemed to be a good compromise between having fewer elements pop in/out and not having indicators merge too quickly.

*Proxies.* Each registered off-screen indicator acts as a proxy for the original element it represents and enables most interactions that can be carried out on the original elements, such as connecting edges, context menus, or tooltips. This behavior is implemented by only changing the appearance of the original element instead of implementing a new element. Consequently, most original interactions remain intact for the indicator and work out of the box.

*Click Listener.* Additionally to all existing interactions, one new interaction was added as well. Clicking on an indicator moves the viewport to the original element. This is done by implementing a new mouse listener, which inherits from the existing *MouseListener* class and listens to the *mouseUp* event on SModel elements. When it is triggered, it is checked whether the element is an off-screen indicator element or not. Next, the number of overlapping indicators is counted. In the case of only one element, the action *CenterAction* is applied, which moves the viewport to center one element (the original SModel element) on the stage. In case of multiple overlaps, the action *FitToScreenAction* is applied, which changes the zoom level and position of the viewport to a point in which all SModel elements that the overlapping indicator represents are visible. Furthermore, it applies a *Select* action, which selects all original elements represented by the indicators.

*Edges.* Edges are usually represented by arrows pointing from a source to a target element. In the developed prototype, we realized that the arrow is still visible even if the source or target element is off-screen. We automatically point to or from an off-screen element and adjust the exact position and size calculation. These adjustments were necessary as indicators retain their size independently of the zoom level, whereas the model content within the current viewport is scaled.

*Rendering.* Before all indicators can be rendered, they have to be prepared. This preparation consists of ($i$) instantiating indicators for each off-screen element, ($ii$) calculating their position around the viewport's border, ($iii$) identifying overlapping indicators, and ($iv$) calculating the new position of merged indicators. All these tasks are triggered every time the model is rendered.

Figure 5 shows the workflow model in our Off-Screen Element prototype. Figure 5a first shows a completely zoomed-out version of the model as a reference, showing three automated tasks and two manual tasks without any off-screen elements. Figure 5b shows a model with three off-screen tasks in the form of off-screen indicators. The color of the indicators gives information about their type (2x automated, 1x manual). Figure 5c shows the same model

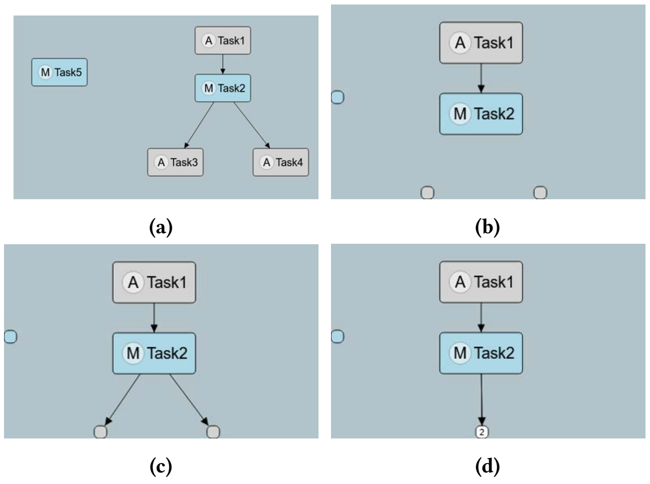

**Figure 5: Screenshots of the Off-Screen Elements prototype.**[13]

and how edges can connect off-screen elements with on-screen elements. Figure 5d shows how multiple off-screen elements were merged into one indicator to reduce cluttering. The number represents the amount of combined off-screen element indicators.

## 5 DISCUSSION AND PERFORMANCE EVALUATION

Both prototypes have been successfully integrated into the Eclipse Graphical Language Server Platform and are fully functional and usable with the current version of the workflow language. The first prototype effectively condenses visible information concerning the current zoom level based on rule definitions (client- and server rules), rule assignment, and rule application. The prototype demonstrates how to integrate a semantic zooming functionality into a GLSP-based client-server architecture by providing a clear separation of concerns. We realized an extendable architecture, e.g., in the form of interfaces to define new rules, which developers can use to add new functionalities easily. During the conceptualization, care was taken to keep the current version of the graphical language server protocol intact and the additions/changes minimal. This was accomplished by adding two new optional actions and their respective responses and one change in the form of an additional optional parameter to the *requestModel* action.

The second prototype adds the visualization of off-screen elements to the client. It extends the already existing functionality of the client implementation to demonstrate how to add cues about off-screen elements at the border of the viewport to maintain a solid mental image of the workspace. Furthermore, it provides new ways of navigating the workspace, decreasing the time required to reach off-screen elements. It also eliminates the need for scrolling and panning actions for other actions, e.g., connecting edges from on-screen to off-screen elements, by making each indicator act as a proxy. Unlike the semantic zoom prototype, the realization operates

[13]Off-Screen elements prototype video: https://www.youtube.com/watch?v=HRq7_olQo08

entirely on the client and does not require any new server functionalities. Its functionality is kept generic to enable its efficient utilization for other modeling languages.

## 5.1 Critical Reflection

While both prototypes are fully functional and usable, they still have some limitations that ideally should be addressed before they can be used on a larger scale. The following sections will go into further details about limitations, potential solutions to them, and other improvements that can be made to both prototypes.

*5.1.1 Semantic Zooming.* The first prototype was not only more sophisticated to conceptualize and develop. It required a slight adjustment of the existing protocol (addition of a zoom level parameter in the *requestModel* action) and two new server-side actions. All these changes are optional and do not have to be called or used, which means that the server can still be used, even with clients that do not actively use semantic zooming.

*Animations.* Each time the discrete LoD is changed, an animation is played that transitions elements from one state to the next. A challenge in the current implementation is that no new user interaction events are processed during this animation. During the animation, the zoom event is not recognized. This is related to an issue requiring adjustments of Sprotty itself[14], as a workaround, we changed the interval in which animations are played to a smaller amount (~200ms). This caused the animation to be played long enough to be still visible but fast enough to not noticeably disrupt zooming events.

*Server Round Trips.* Another limitation is the additional server round trips required during every change in discrete LoD. Because the server is responsible for calculating some values during the rendering process, such as the elements to be shown, the client must request the server before it can re-render the diagram. While this is usually not a problem with discrete LoDs, working with continuously triggered rules would require an extra server round trip every time the user performs a zoom action. Reducing required server round trips, for instance, with preloading detail levels from the server, would further increase the client's responsiveness.

*Dynamic Size Adjustments.* The current implementation automatically adjusts each diagram element's size to its content size. This means that, e.g., whenever a property is made visible by a rule, the parent's size is adjusted to fit the new property. While this is intended behavior, in sporadic cases, it can cause a structural change that destroys the workspace's mental map. A potential solution to this problem would be only to show further elements once there is enough space for them instead of at a fixed zoom level. E.g., when the user zooms in, the newly created space inside model elements is calculated, and new properties are added if they fit into this space. The problem here is that, in the current GLSP implementation, the client cannot calculate the correct sizes for each element because part of that responsibility is also on the server-side. Furthermore, dynamic adjustments could cause two tasks of the same type to add a property at different zoom levels because of differences in the value to render.

[14]https://github.com/eclipse/sprotty/issues/1

*5.1.2 Visualizing Off-screen Elements.* The second prototype is only implemented on the client, which raises the question of whether the server is supposed to be involved in visualizing off-screen indicators. On the one hand, indicators are not directly part of the model, do not have to be persisted, and are directly dependent on information only the client has (e.g., position and bounds of the viewport). All of these reasons speak for implementation on the client-side. On the other hand, being able to control the visualization of off-screen elements from the server speaks in favor of the goal of LSP, which is to prevent having to implement the same functionality multiple times for different clients.

With the current workflow of GLSP, it is hard to shift the entire feature to the server-side for primarily two reasons: ($i$) the server does not have the necessary information to determine the position of indicators; ($ii$) the positions of all indicators have to be adjusted during all zooming or panning events. If the server is responsible for positioning indicators, this would require a server round trip during and after every event.

We believe, ideally, off-screen element functionality should be distributed among the server and the client in GLSP. The logic determining the position and size of indicators should be kept on the client while the server should be in charge of multiple configurational matters. Examples are: which elements should have indicators, CSS classes of different indicators, and logic about when to merge which indicators. All clients can then be implemented to understand the configuration, which is supplied by the server.

*Additional Configuration Parameter.* Another improvement can be made by adding configuration parameters to the client. Currently, the only parameter is the value of *OVERLAP_SIZE_MULTIPLIER*, which determines how close elements have to be to get merged. Another important parameter that is not implemented currently is the maximum distance to the center of the viewport that elements can have before their indicators are not rendered anymore (cf. area of influence [12]). Elements that are too far away or have no relationship with the currently visible elements are often not relevant, only decrease the performance, clutter the workspace, and could be omitted.

*Identification of Indicators.* Knowing which indicator represents which model element can often be complex but is required to utilize this prototype efficiently. Currently, there exist three parameters of an indicator that give information about the model it represents: ($i$) its position, ($ii$) its color, and ($iii$) its edges. This is often not enough and can be further improved, e.g., with ideas given in [12] like a stacking effect for multiple indicators at the exact location or adding specialized interactions for indicators like tooltips, which show the full name of all represented elements.

## 5.2 Performance

For both prototypes, we conducted experiments to evaluate how the performance changes with an increasing number of model elements and indicators. The experiments involved simple zooming and panning interactions on a diagram that consisted of a predefined number of tasks, edges, and indicators. We then measured at what point we recognized a noticeable delay in these interactions. As a reference point, the original implementation of the original

Table 1: Performance evaluation results

| | Elements | | | Render Time | |
|---|---|---|---|---|---|
| | Tasks | Edges | Indicators | Zooming | Panning |
| Original | 10 | 10 | - | ~0.4ms | ~0.4ms |
| | 100 | 100 | - | ~3ms | ~3ms |
| | 500 | 500 | - | ~18ms | ~18ms |
| Sem. Zoom | 10 | 10 | - | ~0.8ms | ~0.8ms |
| | 100 | 100 | - | ~9ms | ~7ms |
| | 500 | 500 | - | ~44ms | ~41ms |
| Off-Screen | 10 | 10 | 0 | ~0.5ms | ~0.5ms |
| | 100 | 100 | 0 | ~4ms | ~4ms |
| | 100 | 100 | 99 | ~12ms | ~10ms |
| | 500 | 500 | 0 | ~27ms | ~26ms |
| | 500 | 500 | 499 | ~76ms | ~50ms |

workflow language server has been taken. Furthermore, the rendering time has been measured for each iteration. The results of the experiments are summarized in Table 1. It has been conducted on the following environment: Firefox 103.0 on Windows 10 with AMD Ryzen 5 5600X and 16GB RAM.

The zooming and panning events showed similar rendering times throughout the experiments. Furthermore, the measured times of both prototypes were very similar and roughly twice as high as those of the original implementation. This can be attributed to the additional logic executed during each iteration of the rendering function in both prototypes. It is to say, while the performance aspect was not wholly neglected during the development of the prototypes, good performance was not a primary focus. For this reason, the performance can probably be improved in many parts of the prototypes, especially by caching calculations and reusing them.

For the subjective evaluation, slight unresponsiveness could be felt during the test execution with 100 Tasks. This delay was barely noticeable and could only be felt when a direct comparison to a diagram with fewer tasks was given. During the test execution with 500 Tasks, the delay became noticeable. This could be felt even more during the execution of prototype-specific functionality. For the semantic zoom prototype during the switch between discrete LoDs and for the Off-Screen Elements prototype while displaying many indicators. Especially the switch between discrete LoDs added a noticeably long delay to the interaction, which is not reflected in the rendering times. It can probably be attributed to the additional server round trip that has to be made in the semantic zoom prototype. Besides, the additional prototype-specific logic also plays a significant role in the responsiveness differences.

We intend both prototypes to be used in combination with models created by humans (see definition of conceptual modeling by Mylopoulos [28]). With many entities, a model often becomes too complex, unmanageable, and difficult to understand [27] – calling for modularization [3, 15]. For this reason, we do not expect the number of model elements to go as high as 500 or even 100. The performance experiments were still beneficial in learning about the current scalability of the prototypes.

## 6 CONCLUDING REMARKS

This paper provided concepts and prototypical implementations to realize advanced visualization and interaction features with the Eclipse Graphical Language Server Platform. While conceptualizing a semantic zoom and an off-screen elements prototype, a focus was placed on keeping it generic and extensible.

The main logic of the prototype implementations can be reused in other language servers based on Eclipse GLSP without much effort. What cannot be avoided is the inherently required language-specific configuration and adaptation to new graphical elements that may be needed in other languages. We believe the given concepts provide a good description for integrating the chosen features into GLSP-based modeling tools. Furthermore, the implementations of both concepts prove their validity and provide a solid foundation for further additions or improvements in future works.

Our performance evaluation shows that for conceptual models of realistic size, i.e., models created and used by humans, semantic zoom and off-screen elements work smoothly. The implementations we developed in this research are open source[15], and we aim to contribute them to future releases of GLSP to make them applicable for any modeling tools developed on the GLSP platform. Moreover, we aim to conduct empirical research to learn more about the perceived usefulness and the ease of use of semantic zoom and off-screen elements for modelers.

## ACKNOWLEDGMENTS

This research has been partly funded by the Austrian Research Promotion Agency (FFG) via the Austrian Competence Center for Digital Production (CDP) under the contract number 854187.

## REFERENCES

[1] Lyn Bartram, Albert Ho, John Dill, and Frank Henigman. 1995. The continuous zoom: A constrained fisheye technique for viewing and navigating large information spaces. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*. 207–215.

[2] Patrick Baudisch and Ruth Rosenholtz. 2003. Halo: a technique for visualizing off-screen objects. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 481–488.

[3] Dominik Bork, Antonio Garmendia, and Manuel Wimmer. 2020. Towards a Multi-Objective Modularization Approach for Entity-Relationship Models. In *ER Forum, Demo and Posters 2020 co-located with 39th International Conference on Conceptual Modeling (ER 2020), Vienna, Austria, November 3-6, 2020 (CEUR Workshop Proceedings, Vol. 2716)*, Judith Michael and Victoria Torres (Eds.). CEUR-WS.org, 45–58.

[4] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. 2018. Systematic analysis and evaluation of visual conceptual modeling language notations. In *2018 12th International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–11.

[5] Dominik Bork and Ben Roelens. 2021. A technique for evaluating and improving the semantic transparency of modeling language notations. *Softw. Syst. Model.* 20, 4 (2021), 939–963.

[6] Hendrik Bünder. 2019. Decoupling Language and Editor-The Impact of the Language Server Protocol on Textual Domain-Specific Languages.. In *MODELSWARD*. 129–140.

[7] Giuliano De Carlo, Philip Langer, and Dominik Bork. 2022. Rethinking Model Representation - A Taxonomy of Advanced Information Visualization in Conceptual Modeling. In *International Conference on Conceptual Modeling (ER'22)*.

[15] latest version: https://github.com/glsp-extensions this paper's version permanently via: https://doi.org/10.5281/zenodo.7007921

[8] William C Donelson. 1978. Spatial management of information. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. 203–209.

[9] Dustin Dunsmuir. 2009. Selective Semantic Zoom of a Document Collection. *Available at,(Oct. 30, 2009)* (2009), 1–9.

[10] Eclipse Foundation. [n. d.]. Eclipse Graphical Language Server Platform. https://github.com/eclipse-glsp/glsp. Accessed: 10.05.2022.

[11] Ulrich Frank, Stefan Strecker, Peter Fettke, Jan Vom Brocke, Jörg Becker, and Elmar Sinz. 2014. The research field modeling business information systems. *Bus. Inf. Syst. Eng.* 6, 1 (2014), 39–43.

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

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

[14] George W Furnas. 1986. Generalized fisheye views. *Acm Sigchi Bulletin* 17, 4 (1986), 16–23.

[15] Giancarlo Guizzardi, Tiago Prince Sales, João Paulo A. Almeida, and Geert Poels. 2021. Automated conceptual model clustering: a relator-centric approach. *Software and Systems Modeling* (2021). https://doi.org/10.1007/s10270-021-00919-5

[16] Jens Gulden. 2016. Recommendations for Data Visualizations Based on Gestalt Patterns. In *International Conference on Enterprise Systems*, Gang Li and Yale Yu (Eds.). 168–177.

[17] Jens Gulden, Hajo A Reijers, J Grabis, and K Sandkuhl. 2015. Toward Advanced Visualization Techniques for Conceptual Modeling.. In *CAiSE Forum*. Citeseer, 33–40.

[18] Sean Gustafson, Patrick Baudisch, Carl Gutwin, and Pourang Irani. 2008. Wedge: clutter-free visualization of off-screen locations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 787–796.

[19] Sean G Gustafson and Pourang P Irani. 2007. Comparing visualizations for tracking off-screen moving targets. In *Extended Abstracts on Human Factors in Computing Systems*. 2399–2404.

[20] Tan Kim Heok and Daut Daman. 2004. A review on level of detail. In *Proceedings. International Conference on Computer Graphics, Imaging and Visualization, 2004. CGIV 2004*. IEEE, 70–75.

[21] Takeo Igarashi and Ken Hinckley. 2000. Speed-dependent automatic zooming for browsing large documents. In *ACM symposium on User interface software and technology*. 139–148.

[22] Huzefa Kagdi and Jonathan I Maletic. 2007. Onion graphs for focus+ context views of UML class diagrams. In *Int. Workshop on Visualizing Software for Understanding and Analysis*. 80–87.

[23] Banda KalyanaChakravarthy. 2008. Visualizing the MPI Programs: Using Continuous Semantic Zooming. (2008).

[24] A Chris Long, Brad Myers, Juan Casares, Scott Stevens, and Albert Corbett. 2004. Video Editing Using Lenses and Semantic Zooming. (2004).

[25] microsoftlspimpl [n. d.]. Microsoft language server protocol implementations. https://microsoft.github.io/language-server-protocol/implementors/servers/. Accessed: 23.04.2021.

[26] microsoftlspspec [n. d.]. Microsoft language server protocol specification. https://microsoft.github.io/language-server-protocol/specifications/specification-current/. Accessed: 23.04.2021.

[27] Daniel Moody. 1997. A multi-level architecture for representing enterprise data models. In *International Conference on Conceptual Modeling*. Springer, 184–197.

[28] John Mylopoulos. 1992. Conceptual modelling and Telos. *Conceptual modelling, databases, and CASE: An integrated view of information system development* (1992), 49–68.

[29] Tom Owen, George Buchanan, Parisa Eslambochilar, and Fernando Loizides. 2010. Supporting early document navigation with semantic zooming. In *International Conference on Asian Digital Libraries*. Springer, 168–178.

[30] Ken Perlin and David Fox. 1993. Pad: an alternative approach to the computer interface. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. 57–64.

[31] Parsa Pourali and Joanne M. Atlee. 2018. An Empirical Investigation to Understand the Difficulties and Challenges of Software Modellers When Using Modelling Tools. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018*, Andrzej Wasowski, Richard F. Paige, and Øystein Haugen (Eds.). ACM, 224–234. https://doi.org/10.1145/3239372.3239400

[32] 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, 9–9.

[33] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. 2018. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. 370–380.

[34] Ben Roelens and Dominik Bork. 2020. An evaluation of the intuitiveness of the PGA modeling language notation. In *Enterprise, Business-Process and Information Systems Modeling*. Springer, 395–410.

[35] Kurt Sandkuhl, Hans-Georg Fill, Stijn Hoppenbrouwers, John Krogstie, Florian Matthes, Andreas Opdahl, Gerhard Schwabe, Ömer Uludag, and Robert Winter. 2018. From Expert Discipline to Common Practice: A Vision and Research Agenda for Extending the Reach of Enterprise Modeling. *Bus. Inf. Syst. Eng.* 60, 1 (2018), 69–80.

[36] George Schussel. 1995. Client/server past, present, and future. *Formerly Available URL: http://news.dci.com/geos/dbsejava.htm* (1995).

[37] Michael Stengel, Mathias Frisch, Sven Apel, Janet Feigenspan, Christian Kästner, and Raimund Dachselt. 2011. View infinity: a zoomable interface for feature-oriented software development. In *Proceedings of the 33rd International Conference on Software Engineering*. 1031–1033.

[38] Christian Tominski, James Abello, Frank Van Ham, and Heidrun Schumann. 2006. Fisheye tree views and lenses for graph visualization. In *Tenth International Conference on Information Visualisation (IV'06)*. IEEE, 17–24.

[39] YoungSeok Yoon and Brad A Myers. 2015. Semantic zooming of code change history. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 95–99.

[40] Polle T Zellweger, Jock D Mackinlay, Lance Good, Mark Stefik, and Patrick Baudisch. 2003. City lights: contextual views in minimal space. In *CHI'03 extended abstracts on Human factors in computing systems*. 838–839.