# Graphical Projectional Editing in Gentleman

Aurélien Ducoin
DIRO, Université de Montréal
Montréal, Canada
aurelien.ducoin@umontreal.ca

Eugene Syriani
DIRO, Université de Montréal
Montréal, Canada
syriani@iro.umontreal.ca

## ABSTRACT

Graphical modeling languages require proper management of position, size, and layout. Most modeling editors lack automated support to manage these graphical concrete syntax properties. It is a time-consuming effort that affects the understandability of the model. Projectional editors prevent end-users from modifying the concrete syntax so they can focus on the modeling task. However, while they offer multiple notations, these editors lack support for graphical languages. In this paper, we present a projectional editor for graphical languages. Our implementation extends the Gentleman editor generator with multiple layouts and interaction-oriented components to generate web editors. A demonstration of the tool is available at https://youtu.be/wd00pRAHxsU.

## KEYWORDS

model-driven engineering, projectional editing, graphical concrete syntax, domain-specific language

**ACM Reference Format:**
Aurélien Ducoin and Eugene Syriani. 2022. Graphical Projectional Editing in Gentleman. In *Proceedings of Tools & Demonstrations — MODELS'22 (MODELS'22).* ACM, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.1145/3550356.3559092

## 1 INTRODUCTION

In model-driven engineering, most graphical language workbenches [3] rely on domain-specific syntax-directed visual editors, like AToMPM [8], MetaEdit+ [4], and WebGME [6]. The graphical models built have precise semantics and must be syntactically correct to process them automatically with code generation, model transformation, or model analysis. Thus, users cannot create syntactically illegal models based on the metamodel and static semantics. However, managing the inherent complexity of graphical representations still hinders users when they should be focused on building their model for the problem at hand. For example, managing edge-crossing, creating a mental map, and organizing the positions of the element in the layout are time-consuming activities that have no impact on the meaning of the model. Automatic layout features can be helpful in the presentation of the final model but not optimal during its development. However, since language workbenches are designed agnostically from the domain-specific languages (DSL) they can generate, most automated styling and layout are generic. The concrete syntax (layout, positioning, style) of a model is utterly important and significantly impacts its understandability. For example, graph-based languages (e.g., UML class diagrams, Statecharts) must group and keep the strongly connected elements close for better readability. Tree-based structures (e.g., family tree, organization chart) must position nodes and leaves in a top-down manner. There are also good practices in the pragmatic use of a modeling language that imposes a standard way of presenting model elements [1].

Projectional editing tries to fix these issues for textual languages. For a given DSL, projectional editors predefine the concrete syntax and let users only edit the abstract syntax of the models. Thus, users are presented with a projection of the concepts which cannot be altered. The projectional editors still active today are Jetbrains MPS [10] and Gentleman [5]. The MPS editor focuses on textual models and programs by partitioning the editor into cells that can hold symbols. Gentleman takes advantage of web technology to have more flexible representations using containers that can hold HTML elements. Although they both support annotations and graphical symbols, they are not suited for graphical concrete syntax since the position of projections is not always as predictable for graph-based syntax as for textual syntax. mbeddr[1] extends MPS and supports graphical notations. However, users can modify concrete syntax elements directly, such as size and position, which does not make it projectional anymore. Other graphical visualization tools, like UMLet [2], do not permit editing the graphical model directly. Instead, users edit the model in a textual syntax, and the diagram is updated accordingly by automatically placing and sizing elements. Hence, graphical projections require a different specification.

**Therefore, we propose a novel category of modeling editors that are domain-specific, graphical, and projectional.** In these editors, users can edit the abstract syntax of their model directly on the canvas, and projections automatically render the visuals in the appropriate layout. The layout is declared in the projection using algorithms that focus on models that can evolve with new elements added or deleted. In this paper, we present a graphical projectional editor generator implemented as an extension of Gentleman. It offers a wide choice of representations and interactions in a graphical and projectional environment while letting the user concentrate on the modeling task without worrying about presentation concerns.

## 2 PROJECTIONAL EDITING

We first briefly present the projectional editor generator Gentleman and then discuss the requirements for graphical projections.

### 2.1 Overview of Gentleman

Gentleman is a web-based projectional editor generator [5]. It uses web technology to customize projections with CSS properties to organize layouts or style elements (e.g., color, font, dimensions). To generate editors, the language engineer has to create a model defining the concepts and another model for their projections.

*MODELS'22, October 23-28, 2022, Montréal, Canada*
2022. ACM ISBN 978-1-4503-8104-8/21/03…$15.00
https://doi.org/10.1145/3550356.3559092

[1] http://mbeddr.com/platform.html

The concept model describes the metamodel of a language. It contains *primitive concepts* (string, number, boolean, reference, set) and *user-defined concepts*. Concepts can be composed with each other by attributes. Attributes are identified by a name and typed by a concept. Inheritance and extension are respectively realized with *prototype* and *derivative concepts*.

The projection model groups the projections of the different concepts in the concept model. A projection is defined for one concept. *Fields* are interaction elements that are bound to the value of primitive concepts. They focus on creation, deletion, selection, and value modification. *Static* elements are not editable, such as labels, icons, and buttons. *Layouts* enable language engineers to customize projections with multiple fields and statics, setting the general organization of the concept elements to be rendered. In addition, language engineers can create *style rules* to define CSS properties that will be shared between multiple projections. Since a concept may have different projections in different contexts, users can switch between the projections available for that concept. Projections can be reused via *tags* (reference labels) or by defining template projections. During the development of a projection, the language engineer may preview the editor to be output.

Once completed, the language engineer builds her DSL, which produces the JSON files for concepts, projections, and, optionally, the configuration of the editor. Gentleman can be integrated into web pages using a simple HTML tag, with a JavaScript script, or via its API. End-users (i.e., users of this DSL) create and edit their models in the Gentleman editor preloaded with the above-mentioned JSON files. It is possible to customize the concepts and projections to define concepts and projections since Gentleman is bootstrapped.

Gentleman creates container-based projections (similar to nesting in JSON) that are well-suited to project concepts in HTML. However, this is a limiting factor for graphical projections that require more diversified designs.

## 2.2 Requirements for graphical and projectional

Projectional editors highly constrain the user interactions to guarantee that the current model is always syntactically correct. The only permissible user interactions must solely affect the underlying abstract syntax graph (ASG) of the currently-loaded model [9]. Although the user interacts with a projection of the concept, the projection handler delegates the operation to the ASG handler of the concept. Hence, projections must evolve according to the evolution of the ASG. For text and container-based editors (like those supported by MPS and Gentleman), the main evolution of a projection concerns spacing between elements, line breaks or wrapping, and indentation or nesting. These can be handled automatically by the tool. The language engineer has only control over horizontal/vertical flow, tabular organization, and styling of projections. Graphical projections require more aspects to control because it is not possible to predict the evolution of complex graphics. Hence, we outline necessary requirements specific to graphical projectional editing.

*2.2.1* ***Req #1: The layout must automatically resolve the position of an element.*** A graphical projection must be placed on the canvas to render the underlying concept visually. If the concept contains attributes that dictate its coordinates, a mapping between the values and coordinates can be easily resolved. However, this is not the case in general. The position of a projection is often related to the position of other projections. For example, consider a DSL for family trees. If a person has a higher y-coordinate than another person, then the former is an ancestor of the latter. Therefore, if the graphical language requires structures and topology, a collection of layouts must be available to the language engineer who can control their parameters.

*2.2.2* ***Req #2: The entire context of a relation must be resolved before connecting projections.*** In textual languages, connectors are usually implicit in nesting or a reference. Visual connectors typically involve a source and a target. In syntax-directed editing, it is possible to create the target on-the-fly with the connector since the target can be placed anywhere. In graphical projectional editing, Req #1 ensures that the target is placed appropriately, which may be different if it is connected to the source. It is also possible to define a relation as a concept, such as the concept of marriage connecting two people. In this case, both source and target must exist in the ASG of the model before creating the relation.

*2.2.3* ***Req #3: The layout should modify elements size.*** Adding and removing elements in a collection implies resizing the projection of the concept containing them. Since a graphical projection comprises many components (e.g., shapes, connectors, textual labels, style), specifying the size evolution of the projection—and its components—is challenging. In traditional editors, the end-user can manually adapt sizes for better visualization. However, in projectional editing, this type of interaction is not available because it does not affect the ASG of the model. In addition, layout disposition may be related to the size of the projections. For example, augmenting the size of one projection might lead to superposition problems in the layout. Therefore, the layout is best suited to manage the size and guarantee the robustness of the general structure. In addition, communications between the different projections should be efficient enough to determine the impact of such evolution.

*2.2.4* ***Req #4: Visual aesthetics must be considered when designing the layout.*** The language engineer creating a layout for a projection should take visual aesthetics into account. Similar models should be rendered similarly, ideally in a canonical form. The visual projection should follow principles for designing cognitively effective visual notations [7]. For example, minimizing the presence of model elements that may cause misunderstandings, avoiding edge-crossing, ensuring that the model is neither too compact nor too scattered, ensuring the general organization maintains the mental map of the model, and minimizing the presence of buttons to emphasize the model rather than interaction points.

*2.2.5* ***Req #5: Each interaction point shall be designed with its own graphical projection.*** Users who interact with a projectional editor fill in blanks in the ASG or add/remove subgraphs. In text-based and container-based editors, the structure of the ASG is easily identifiable: interaction regions can be highlighted or presented in text boxes, choices can be expanded, and adding/removing elements can be delegated to specific keys or buttons. This is less obvious in a graphical projection because more information is displayed to the user, and multiple shapes, style elements, and positioning may represent a single concept. For example, in a sequence

diagram, the lifeline, activation boxes, and object can all visually project a single concept of object lifeline. Thus, it is harder for the end-user to identify and understand the ASG and decide which action to perform on it. Moreover, relying solely on buttons in a region crowded with interaction points may lead to more confusion. Therefore, it may not be feasible to show all interaction points for a concept in the same projection as this may clutter the overall visualization of the model.

*2.2.6* ***Req #6: Projections that are only designed for interactions must be limited to ensure understandability.*** A consequence of Req #5 implies that if a complex concept is composed of several concepts (via attributes, relations), its projection consists of the projection of each of these concepts combined and organized into one visualization. Since users should only interact with specific parts of the ASG, interaction points (e.g., buttons) must be designed and placed accordingly. They should ensure that visual aesthetics are preserved (Req #4) and enhance the understandability of the ASG to the user.

# 3 IMPLEMENTING GRAPHICAL PROJECTIONS

We now outline some design decisions to implement graphical projections in Gentleman following the previous requirements. Figure 1 illustrates graphical projectional editors for three DSLs.

## 3.1 General guidelines

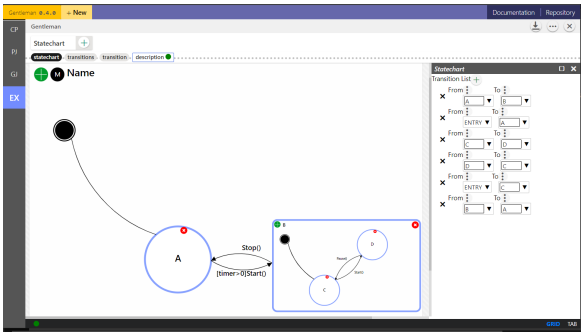
We wanted to maximize the user experience to feel like a graphical modeling tool by focusing most interactions directly on the canvas where the model is presented. We also wanted to reuse the Gentleman editor as much as possible. Since Gentleman relies on HTML/CSS to render projections on the web page, we represent graphics with scalable vector graphics (SVG). Each vector can be modified, styled separately, and injected into the DOM.

The editor renders each container-based projection in a separate `<div>` and automatically organizes them with CSS rules. Since this is not enough for graphics (Req #1), we created a new projection type dedicated to graphical syntax. This way, all concepts can be projected graphically as well. However, container-based projections cannot be reused as-is because their size is adapted to the containing `<div>`, while we need the layout to control it (Req #3). Nevertheless, a given concept may have a textual/container-based and a graphical projection. The end-user can display them simultaneously using a side window for textual interaction.

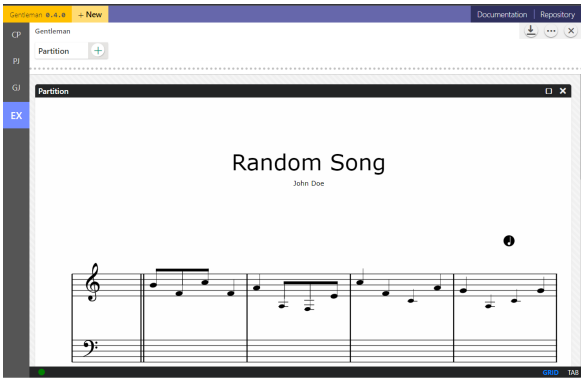
In the new projection type, we extended *fields* and *statics* to be integrated with graphical projections. We designed several generic graphical layouts that can be parameterized for the DSL. To help the language engineer design shapes and vectors, we developed a simple SVG editor to preview and create the shapes. To better parameterize the layouts, she is also provided with a layout simulator to observe the effect of the parameters on a sample graphic.

Since Gentleman is bootstrapped, we created the new fields, connectors, and layouts as new concepts in the graphical projection metamodel. The graphical projection editor is available in the Gentleman distribution[2].

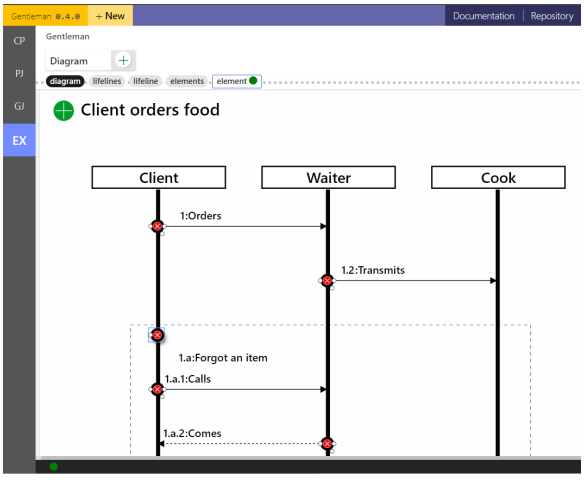
[2]https://github.com/geodes-sms/gentleman/tree/Graphical




(a) Statecharts editor




(b) Music sheet editor




(c) Sequence diagram editor

**Figure 1: Graphical editors in Gentleman in various layouts**

## 3.2 Interacting with graphical projections

Interactions with a graphical projectional editor are very restricted. They typically include adding/removing concepts, setting attribute values, and establishing references to other concepts. We describe how projections can implement these actions while conforming to the requirements.

*3.2.1 Creating and removing elements.* When creating a concept instance in the model, its container is responsible for providing the interaction point, often in the form of a button. If it is a root concept,

the end-user is presented with a button to create the concept directly on the canvas representing the model. Similarly, a button to remove a concept instance is available on the projection of that concept. The button sends position information to the layout to place the element accordingly (Req #1). Buttons can be hidden when the end-user stops interacting with the projection (Req #6).

*3.2.2 Setting values.* Setting a value may target a primitive or user-defined concept. For the former, we reuse Gentleman projections for booleans, numerals, and strings. If the value must be displayed on the graphical element, like a person's name in the family tree, we create a *text field* that annotates the SVG element and can be directly edited. It has an anchor to position the text, a placeholder for the text to be always visible, and can be styled (Req #4).

For the latter, if there is a single possible concept for the value, this corresponds to Section 3.2.1. Otherwise, a *choice field* presents the possible values. It is particularly useful in the case of a prototype concept (similar to an abstract superclass); the projections of all compatible concepts are presented if they share the same tag value. The language engineer can specify maximum rows/columns to present in a tabular form. For example, in the family tree DSL, a sibling may have two potential values for brother and sister.

In some cases, the value of a concept directly impacts a concrete syntax property. For example, in a DSL for music sheets, the value of a note dictates its position on the staff. The *placeholder field* displays all the possible renderings of a concept inside its SVG element. The end-user selects the desired placeholders, which sets a mapping to its value (Req #6).

When a concept has an attribute with multiple predefined values (i.e., constraint enumerates the possible values), relying on multiple choice fields may clutter the projection, hindering the understandability of the model (Req #5). Therefore, the *switch field* allows the end-user to rotate through the values of the attribute. Clicking on the projection modifies the value according to a predefined order. For example, in the music sheet DSL, the tempo of a note is represented with a *switch field*. Clicking on a quarter note will make it half, then whole, then eighth, and then back to quarter.

Changing a value may have an impact on concrete syntax properties. For example, if the person's name is too long, it may be repositioned (Req #1) or resized (Req #2). In general, the modification of a value may affect multiple vectors of the projected SVG. We solved this by introducing *markers* as SVG attributes. When the language engineer wants to specify the resulting changes applied when setting a value, she can identify the targeted elements by specifying the marker. The modifications can either be mapped to a value by selecting properties or contained in the value of the marker.

*3.2.3 Establishing references.* In graphical notations, references are usually rendered as connectors (e.g., arrows). As stated in Req #2, creating a reference requires both the source and target concepts to already exist in the ASG. Also, the layout should limit edge-crossing to improve the understandability of the model (Req #4). mbeddr follows the typical interaction of selecting the connection type and click-and-drag from the source to the target concepts. Gentleman does not support this kind of interaction, which would require generalizing event listeners in the code base and forcing the language engineer to program a specific interaction for each DSL editor.

Therefore, we introduce *projection shadows*. They define a pair of projections, one focusing on the interactions and the other on rendering the connector. Once the source and target values are provided in the first shadow, the second one defines the connector style (e.g., arrow head, solid line) and delegates its position to the layout (Req #1). Since they are two projections of the same concept, modifying the concept affects both shadows.

## 3.3 Layout management

As stated in Req #1–3, layout management is essential in graphical projections, as they control the position and size of the SVG elements to display. We describe the three main layout concepts that we developed in Gentleman. Note that all layouts can be composed with each other to render more complex structures. For example, the music sheet editor (in Figure 1b) is a combination of decoration and pattern layouts.

*3.3.1 Decoration layout.* This layout is useful when the elements it organizes have predefined positions and sizes. It can be used to project any type of concept, typically the root concept of a model acting as the canvas. Background can be provided for aesthetics or meaning (e.g., swim lanes in activity diagrams). The decoration layout receives a set of projections with their absolute/relative size and coordinates. It automatically scales the projections accordingly.

The decoration layout is used for all non-interactive graphical projections, i.e., where no reorganization is needed when interacting with the elements it groups. The music sheet example in Figure 1b uses a decoration layout with predefined dimensions and a specific position for the name of the partition and the staves. Each note has a background shape and specific coordinates. This prevents defining coordinates for each possible value of the prototype concept `Note` in its projection and delegates it directly to the value.

*3.3.2 Pattern Layout.* This layout is useful when the organization of elements has a regular and predictable positioning as more elements are added or removed. It can be used to project a set of concepts (i.e., collections), sharing the same tag and following a pattern. The pattern layout requires the coordinates of the projection of the first element and an offset to place the successive projections, so-called *anchors*. These projections are connected to the pattern so that, when deleted, the layout stays compact by moving every pattern to the previous coordinates (Req #3). When a large number of concepts must be projected on this pattern, the language engineer can set a maximum capacity or a transformation to resize the projections (Req #2). An *anchor template* can also be specified to be reproduced at each anchor, defining a more complex projection.

The pattern layout is used to position each successive note on a staff in Figure 1. It is also used for the sequence diagram editor. Each anchor describes the position of a lifeline. Lifelines also share an anchor template to manage the position of message calls, returns, and fragments. When the next anchor is placed further than the bottom of the lifeline, a transformation increases its height and, conversely, decreases it upon deletion.

*3.3.3 Force Layout.* This layout is useful for projecting graph-like structures, where the primary concern is to avoid the superposition

of elements. It can be used on any type of concept sharing the same tag. The force layout is a common visualization layout to render a network of nodes and edges based on a repelling factor, centers of gravity, distance between elements, and collision detection. For our purposes, the language engineer only needs to specify the distance between two elements linked with a connector and the repelling factor. For this layout, we integrated the D3.js simulator[3] in Gentleman. It is a library for data visualization that generates the coordinates of data points based on constraints. By default, data points are simple coordinates. Thus, we extended them to represent complete projections. Also, since D3 assumes a fixed number of data points that cannot be modified interactively, we extended the simulator with additional functions to handle the addition, deletion, and modification of concepts.

When a concept is added, or its projection is modified (e.g., its size increases due to a value change), the nodes representing the projections of concepts may collide in D3. Therefore, we implemented a collision prevention mechanism by considering the dimensions of projections in the data. In the force layout, we assume one center of gravity for the entire area managed by the layout. Based on the center and size of the layout, we estimate an offset needed to move the colliding projections. However, since this offset does not consider the shape of the projections, the connector connecting them needs to anchor on the border of the shape. Therefore, we estimate the border position and the optimal point of anchor of the shape. For convex shapes (e.g., rectangles, ellipses), we apply geometric computations to identify the closest anchor point to the other shape. For more complex shapes, the language engineer can define a set of anchor point coordinates, and we use the closest one. This ensures the optimal placement of connectors (Req #2).

Nevertheless, the calculation of connectors position may lead to edge-crossing situations (Req #4). As they are curves, if multiple connectors have the same source or target, we increase the curve width to avoid overlapping edges. The resulting curve is then sent to the projection of the connector, represented by an SVG path. To avoid edges from distinct sources and targets crossing, we add a D3 node (invisible to the user) in the middle of each curve. The repelling factor takes care of the rest. The middle node is also used to project decoration on the connector, like text.

D3 simulations call a tick function periodically to update the network. Thus, projections appear to be floating as their coordinates are constantly being updated. This is problematic when addition and deletions occur: the overall organization of the model may significantly change, thus changing the mental map the end-user has of the model (Req #4). Another limitation of using D3 is that the simulation randomly assigns the original coordinates of every node. To overcome this issue, we also store the last coordinates of the model element when it is exported/saved to the JSON file. Thus, the next time the end-user loads the model, all elements will start at the same position and evolve with the ticks.

The Statecharts editor uses a force layout to manage the position of states and transitions in Figure 1. When a new state is created, the editor allows the end-user to choose which concept to add to the model thanks to the choice field (c.f. Section 3.2.2), such as a basic, composite or orthogonal state. Upon selection, the corresponding value is sent to the layout that adds the new data point with its size. In this example, we modeled transitions as concepts with references to the source and target state and various attributes for e.g., triggers and guards. This concept is projected as an arrow, and the attributes are projected in its middle node. Figure 1 shows how different shapes are used in this editor, and we notice no crossing edge and arrows correctly connecting the border of the shapes.

[3]https://d3js.org/

## 4 CONCLUSION

Creating graphical projectional editors means taking more constraints into account. The implementation presented in this paper offers possible tracks to explore to improve the generation of such editors. The layouts allow the creation of graph-like models, but more specific structures can be defined for projectional editors, such as a tree layout. Because projectional editing focuses on interactions, we are working on improving the user experience and diversifying the possible actions.

## REFERENCES


[1] Scott W Ambler. 2005. *The Elements of UML 2.0 Style.* Cambridge University Press.

[2] Martin Auer, T Tschurtschenthaler, and Stefan Biffl. 2003. A flyweight UML modelling tool for software development in heterogeneous environments. In *Euromicro Conference*. 267–272.

[3] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, 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 A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. 2013. The state of the art in language workbenches. In *International Conference on Software Language Engineering (LNCS, Vol. 8225)*. Springer, 197–217.

[4] Steven Kelly, Kalle Lyytinen, and Matti Rossi. 1996. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Conference on Advanced Information Systems Engineering (LNCS, Vol. 1080)*. Springer, 1–21.

[5] Louis-Edouard Lafontant and Eugene Syriani. 2020. Gentleman: a light-weight web-based projectional editor generator. In *Model Driven Engineering Languages and Systems: Companion Proceedings*. 1–5.

[6] Miklós Maróti, Tamás Kecskés, Róbert Kereskényi, Brian Broll, Péter Völgyesi, László Jurácz, Tihamér Levendovszky, and Ákos Lédeczi. 2014. Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. In *Multi-Paradigm Modeling workshop*, Vol. 1237. CEUR-WS.org, 41–60.

[7] Daniel Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35, 6 (2009), 756–779.

[8] E. Syriani, H. Vangheluwe, R. Mannadiar, C. Hansen, S. Van Mierlo, and H. Ergin. 2013. AToMPM: A Web-based Modeling Environment. In *MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition*, Vol. 1115. CEUR-WS.org, 21–25.

[9] Markus Voelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. 2013. mbeddr: Instantiating a language workbench in the embedded software domain. *Automated Software Engineering* 20, 3 (2013), 339–390.

[10] M. Voelter, J. Siegmund, T. Berger, and B. Kolb. 2014. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering (LNCS, Vol. 8706)*. Springer, 41–61.