Université de Montréal

Generating graphical and projectional editors

par

Aurélien Ducoin

Département d'informatique et de recherche opérationelle
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

September 16, 2022

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Generating graphical and projectional editors

présenté par

Aurélien Ducoin

a été évalué par un jury composé des personnes suivantes :

Pierre Poulin
(président-rapporteur)

Eugene Syriani
(directeur de recherche)

Michalis Famelis
(membre du jury)

Résumé

En ingénierie dirigée par les modèles, les langages spécifiques au domaine (DSL) offrent des notations adaptées à un domaine précis pour représenter ses différents concepts. De nombreux outils permettent la définition de DSLs en explicitant les relations entre un concept et ses représentations. En fonction de la sémantique du domaine, l'ingénieur du langage peut choisir entre des notations textuelles ou graphiques. Les langages de modélisation graphique nécessitent une gestion de la position, la taille et la disposition des éléments visuels afin de maximiser leur expressivité visuelle. La plupart des éditeurs de modélisation manquent de support automatique pour gérer ces propriétés de la syntaxe concrète. Les éditeurs projectionnels permettent aux utilisateurs de se concentrer sur la conception de leur modèle en limitant les modifications de la syntaxe concrète. Cependant, bien qu'ils offrent de multiples notations, ces éditeurs ne permettent pas la création de langage graphique. Dans ce mémoire, nous proposons une nouvelle approche pour concevoir des éditeurs graphiques et projectionnels. Nous avons créé une extension d'un éditeur projectionnel orienté vers le web, Gentleman, qui nous a permis d'extraire différentes exigences. Au cours du mémoire, nous décrivons leurs impacts sur les projections et proposons des lignes directrices ainsi que des exemples d'implémentation. Comme l'édition projectionnelle demande une gestion spécifique de l'interaction, nous présentons différentes approches pour interagir avec les représentations graphiques utilisant les nouvelles informations disponibles dans les projections. Étant donné que la plupart des exigences se concentrent sur la disposition des projections, nous avons défini plusieurs algorithmes simples de disposition qui couvrent une large gamme de structures pouvant être retrouvées dans un éditeur graphique. Enfin, afin d'évaluer cette approche, nous avons exploré la génération de trois éditeurs graphiques et projectionnels pour différents domaines: les machines d'états, les diagrammes de séquences et les partitions de musique.

Mots-clés: Ingénierie dirigée par les modèles, édition projectionnelle, syntaxe concrète graphique, language spécifique au domaine

Abstract

In model-driven engineering, domain specific-languages (DSL) provide tailored notations towards a specific problem domain to represent its different concepts. Multiple tools allow the definition of DSL by specifying the relations between a concept and its representations. Depending on the semantics of the domain, the language engineer can choose between textual or graphical notations. Graphical modeling languages require proper management of position, size, and layout to maximize their visual expressiveness. 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. During this thesis, we propose a new approach to design graphical and projectional editors. We created an extension of a web-oriented projectional editor, Gentleman, that allowed us to extract different requirements. During the thesis, we describe their impact on the projections and propose guidelines and examples of implementation. Because projectional editing requires specific management of the interaction, we present multiple approaches to interact with the graphical representations, using the new information available in the graphics. Since most of the requirements were focusing on the disposition of the projection, we define multiple simple layout algorithms that cover a large range of structures that can be found in a graphical editor. Finally, we explore the generation of three graphical and projectional editors for different domains: statecharts, sequence diagrams, and music sheet.

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

Contents

List of Tables

List of Figures

List of abbreviations and acronyms

MDE	Model-driven engineering
DSL	Domain-specific language
AST	Abstract syntax tree
ASG	Abstract syntax graph
OCL	Object Constraint Language
UML	Unified Modeling Language
GMF	Graphical Modeling Framework
KIELER	Kiel Integrated Environment for Eclipse Rich Client
IDE	Integrated developement environment
SVG	Scalable Vector Graphics

Remerciements

I want to express my sincere gratitude to my supervisor Prof. Eugene Syriani. I particularly want to thank him for being very supportive and understanding during difficult times. His knowledge and insights during were crucial this thesis.

I also want to thank my colleagues at the GEODES lab for their help. I would especially want to acknowledge Jessie Gallaso-Carbonnel, for all of her advice, and Louis-Edouard Lafontant, for helping me join the Gentleman journey and all the interesting debates we had. I also want to thank my friends that supported me during these two years.

Finally, I want to dedicate this thesis to my father, mother, sister, and brother. You made this experience possible and I cannot wait to come back to you.

Chapter 1

Introduction

1.1. Context

Visual editors assist users to draw semantically meaningful diagrams. Free-form graphical editors (e.g.,VISIO [1], DRAW.IO [2]) offer almost unlimited freedom in the creation and edition of graphical representations. Users can manually draw complex figures using predefined shapes and curves, and combine them. However, editing activities do not take into account the syntactical validity of their effect with respect to the underlying concepts that are represented in the diagram (e.g., users can create an edge without any target in the Visio flowchart diagram). Also, users have to handle a large set of graphical components and interactions that may not always be relevant when creating diagrams for specific formalisms (e.g., DRAW.IO always offers the exact same shapes in its sidebar for all models).

Graphical representations play a crucial role in software engineering. From class diagrams [**1**] to statecharts [**2**], multiple formalisms help developers design or visualize different parts of complex systems. To ensure continual control of the syntactic validity of graphical representations, syntax-directed graphical editors [**3**] perform analysis during the editing activities based on the rules of the language. They use different visual cues to inform users of potential errors in the graphical model. Limiting the graphical representations and interaction enables users to focus more on the semantics of the diagrams rather than on their creation.

As the complexity of software keeps on increasing, model-driven engineering (MDE) proposes to use abstractions of a system, models, to help the development process [**4**]. Considering that each domain is related to specific semantics, domain-specific languages (DSL) allow the creation of dedicated modeling languages [**5**]. By defining the abstract syntax (the concepts) and the concrete syntax (their representation), a DSL offers tailored notations to

[1] https://www.microsoft.com/fr-ca/microsoft-365/visio/flowchart-software/
[2] https://app.diagrams.net/

help domain-expert design models for a specific problem. Using language workbench, language engineers can then generate IDEs dedicated to a DSL [**6**]. For a graphical concrete syntax, they usually create visual representations for the concepts that they can then add to a canvas to design the model. The resulting editors usually use a syntax-directed approach to prevent syntactical errors, such as MetatEdit+[**7**] and AToMPM[**8**]. However, visual information that is not considered relevant to the semantics is ignored. Managing edge-crossing, creating a mental map, and organizing the positions of the elements in the layout are time-consuming activities that have no impact on the meaning of the model. Manually adapting these different visual properties still hinders users when they should be focusing on building their models for the problem at hand.

Projectional editing is a promising approach to get rid of these concerns by applying constraints on the available interactions. In projectional editors, users directly interact with the abstract syntax using predefined representations called projections [**9**]. Modifications of concepts are limited to syntactical correctness, so the concrete syntax does not need to be parsed. This process enables projectional editors to offer more freedom of notations.

1.2. Problem statement and thesis proposal

Very few frameworks allow the definition and generation of domain-specific projectional editors. Mostly two remain active today. MPS [**10**] is an open-source language workbench developed by JetBrains. Using various DSLs, MPS allows the generation of projectional editors using textual projections. The language workbench also offers code generation and a large diversity of notations [**11**]. As MPS is very heavy-weight, Gentleman offers a web-based editor. Gentleman has its own structure for defining concepts and mapping them to their representations. Using technologies of the web, the language engineer can create, organize, and style container-based projections with specific layouts and interaction-oriented components. The resulting editors can then easily be integrated into other web applications. However, none of these solutions only support limited graphics. When considering for example sequence diagrams or family trees, a graphical concrete syntax can be preferable. Gentleman only supports static images as decorators and MPS is still working on a concrete definition of projectional editing with graphical notations.

Therefore, this thesis proposes a novel category of modeling editors that are domain-specific, graphical, and projectional. To define and generate this new type of editors, we have created an extension of Gentleman that focuses on graphical projections. From this extension, we have extracted requirements and guidelines that are necessary for graphical and projectional editors. Examples of implementation are shown during the thesis to support our claims.

1.3. Contributions

This thesis aims to help the creation of graphical and projectional editors by defining different requirements and guidelines. To support our claims, we created an extension of Gentleman, an open-source web-based projectional editor, that focuses on graphical projections. The contributions are:

- Specific requirements to create graphical and projectional editors.
- Different structures to interact with the projections and layouts to manage their disposition.
- A usable extension of Gentleman to generate and interact with graphical and projectional editors.
- Three application examples that cover different challenges when considering a projectional graphical concrete syntax.

1.4. Outline

This thesis is organized as follows. We start by defining some of the notions necessary to the understanding of our contributions and discuss related work in Chapter 2. In Chapter-3, we focus on the definition of graphical and projectional editors. We start by presenting our extension of Gentleman and the different interaction-oriented structures we created to extract basic requirements. Chapter 4 focuses on the creation of edges and connectors in a projectional editor. After defining the different components for the interaction, Chapter 5 discusses the problem of layout management. We first introduce the creation of graph-based layouts before explaining the necessity of other types of layouts. These different layouts are presented in Chapter 6, with examples of implementation. Finally, Chapter 7 focuses on three editors we generated with Gentleman to support our contribution before concluding in Chapter 8.

Chapter 2

Background and state of the art

In this chapter, we introduce the notions relevant to this thesis. We present MDE and its relation to graphical editing. We then explain the purpose of projectional editing to motivate the work presented in this thesis.

2.1. Model driven engineering

As software becomes more complex, the MDE paradigm focuses on models to support the development process [**4**]. Models are artifacts that represent parts of a system with a certain level of abstraction that encourage a better understanding of the underlying structure and behavior. Each domain has specific semantics that require a different consideration when creating new abstractions [**12**]. To allow domain experts to understand different models, DSLs offer tailored concepts and notations that focus on a specific problem domain.

2.1.1. Domain specific language

A DSL helps its end-users (i.e., experts in that domain) focus on a description of the domain without considering details relative to the implementation [**5**]. Using the different structures of a DSL, they can construct domain-specific models that conform to the rules of the language. Transformations can then be applied to a model to translate into another language or for code generation [**13**]. The definition of a DSL requires a syntax described with two main components: the abstract and concrete syntax.

Abstract syntax. The abstract syntax describes the different concepts of a domain and their relations with the possibility of additional constraints relative to the semantics. Defining the abstract syntax of a language can be done using a metamodel. A metamodel is composed of concepts with attributes and associations that can be represented using the class diagram formalism of the Unified Modeling Language (UML) [**14**]. Multiple tools allow the definition of metamodels by using either textual (PlantUML [**15**], KM3

[16]) or graphical (Eugenia [17], AToMPM [8]) notations. To create the metamodel, the language engineer needs to create a mapping between the concepts and the semantic domain. Semantics regroup the meaning of the different concepts in the domain. To avoid any confusion, each concept must be mapped to a unique definition in the semantics. Additional static semantics can be added to a metamodel by defining constraints using, for example, the Object Constraint Language [18].

Concrete syntax. The concrete syntax is composed of different notations used to represent the different elements of the model [1]. A textual concrete syntax uses grammar to define a structured representation of the abstract syntax. The grammar applies constraints on the text to specify valid inputs or static representations like linebreaks or spaces. The grammar allows the generation of a parser that can ensure that a model is syntactically correct. Frameworks like Xtext [19] or TCS [20], for example, use a specific approach to create the mapping between a representation and its concept [21]. These textual notations usually rely on auto-completion to suggest values to the end-user. In this thesis, we focus on graphical concrete syntax.

2.1.2. Graphical concrete syntax

In a graphical concrete syntax, concepts are represented with different shapes and icons. Typically, these representations are often compared to a diagram with nodes and edges [22]. Creating a graphical concrete syntax requires creating a mapping between shapes and the concepts contained in the abstract syntax. Different approaches can be considered when creating this mapping.

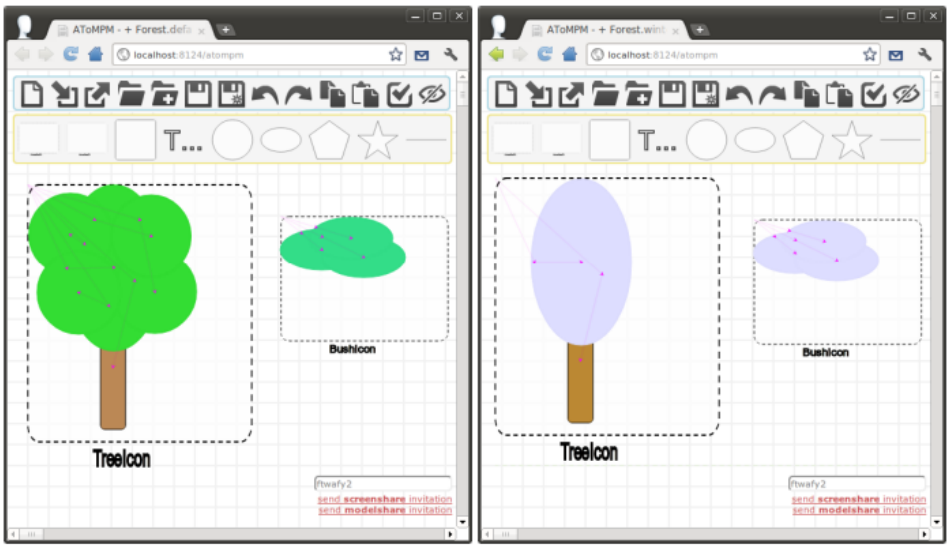


Figure 2.1. Definition of a concrete syntax on AToMPM.

Mapping-based. In the mapping-based approach, the relation between the concepts and their representation has to be explicitly defined. Figure 2.1 shows the definition of a graphical concrete syntax in AToMPM as demonstrated in [**23**]. Here the abstract syntax is composed of two concepts: *Tree* and *Bush*. To create a representation for each concept, the language engineer drags an *Icon* from the toolbar to the canvas. Different shapes can be drawn and styled using Scalable Vector Graphics (SVG) in the icon to create the graphical representation. Relations can be represented as *Links* and decorated with SVG attributes. In addition, the graphical syntax also creates a dedicated toolbar for the DSL to customize the resulting editing environment. As seen in the figure, different concrete syntax can be created for a single abstract syntax for different users of the DSL.

```
1  @namespace(uri="scl", prefix="scl") @emf.gen(basePackage="org.eclipse.epsilon.eugenia
2  .examples")
3  package scl;
4
5  @gmf.diagram
6  @gmf.node(label="name", color="232,232,232")
7  class Component {
8      attr String name;
9      @emf.gen(propertyMultiline="true")
10     attr String description;
11     @gmf.compartment(layout="free")
12     val Component[*] subcomponents;
13     @gmf.affixed
14     val Port[*] ports;
15 }
16
17 @gmf.link(source="from", target="to", label="name", target.decoration="arrow")
18 class Connector {
19     attr String name;
20     ref Port#outgoing from;
21     ref Port#incoming to;
22 }
23
24 @gmf.node(figure="ellipse", size="15,15", label.icon="false",
25     label.placement="external", label="name")
26     class Port {
27     attr String name;
28     val Connector#from outgoing;
29     ref Connector#to incoming;
30 }
```

Listing 1. Annotation in an Ecore metamodel.

Annotation-based. Frameworks such as Eugenia [**17**] propose to specify the graphical representations directly in the metamodel with annotations and model transformations. Listing 1 describes a metamodel for a Simple Component-Connector Language. Each model element has specific annotations that describe its graphical representation. The language engineer can choose the figure and modify parameters such as the color or dimensions. Links directly refer to attributes of a concept to define their source and target. The ***@gmf.diagram*** annotation allows the creation of a canvas and can be attached to the root concept of the metamodel. This annotation-based approach centralizes language information in a single artifact.

```
1  public PictogramElement add(IAddContext context) {
2
3      EClass addedClass = (EClass) context.getNewObject();
4      Diagram targetDiagram = (Diagram) context.getTargetContainer();
5      IPeCreateService peCreateService = Graphiti.getPeCreateService();
6
7      ContainerShape containerShape =
8      peCreateService.createContainerShape(targetDiagram, true);
9
10     IGaService gaService = Graphiti.getGaService();
11
12     RoundedRectangle roundedRectangle =
13     gaService.createRoundedRectangle(containerShape, 5, 5);
14
15     link(containerShape, addedClass);
16
17     return containerShape;
18 }
```

Listing 2. Definition of the add feature in Graphiti.

API-based. The API-based approach offers a dedicated library that implements an API to describe the graphical representations. In Graphiti [**24**], creating the graphic syntax requires an implementation of the *DiagramTypeAgent*. This interface manages the creation of the visual representations and links them to a model element. To allow the evolution of the information in the model, the developer has to create *Features* for the different editing activities. Listing 2 describes the *add* feature for an *EClass* in a specific editor. After loading the diagram and the class connected to the feature in Lines 3–4, the function creates a container for the graphics. Shapes can be drawn and added to the container (Line-12). Before adding the container to the editor, the mapping between the concept and its representations is explicitly defined in Line 15.

The choice between a graphical or a textual concrete syntax is strongly related to the customs of the domain. Graphical notations have been considered superior for some time but research also has shown that they lack the expressiveness of the text [**25**]. Solutions propose to combine both representations [**26**] to take advantage of the two notations.

2.2. Graphical editors

To create and interact with models, a DSL also requires an editing environment. Tools like AToMPM [**8**], MetaEdit+ [**7**], and Sirius [**27**] enable generating an editor for a DSL with a graphical concrete syntax. Editing activities differ from text-oriented editors as they offer more degrees of freedom, such as the positioning or sizing of the graphics. In this thesis, we make the distinction between two types of users. The *language-engineer* uses a tool to generate an editing environment. The *end-user* can then interact with the resulting editor.

2.2.1. Graphical editing

Sirius [**27**] is a framework based on GMF[**28**] that focuses on rapidity and productivity to create graphical editors for a DSM. Generated editors are similar to the one visible in Figure 2.2.

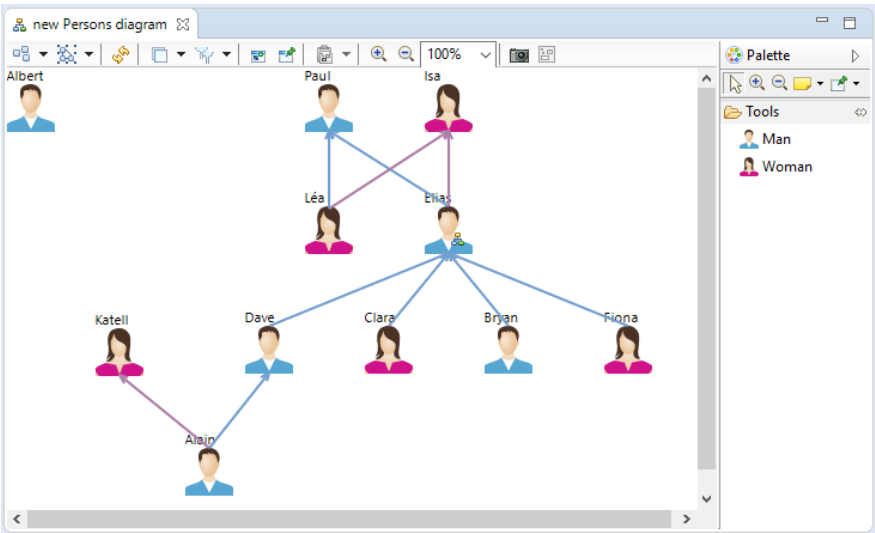


Figure 2.2. Generated editor from a Sirus tutorial.

With Sirius, the language-engineer can define representations and a toolbar with specific actions. The internal representation of the editor uses a tree structure. The editor in Figure 2.2 represents a family and the different relations between people.[1] The toolbar to the right of the editor displays the different concepts that can be created. To add a new *Woman* to the family, the end-user clicks on the button and the concept can be rendered after putting

[1]https://wiki.eclipse.org/Sirius/Tutorials/StarterTutorial

the cursor on the canvas and performing a second click. Size and position can be adapted by interacting with the border of a person.

Links are represented as *Relation Based Edges*. To create a new type of relation, the language-engineer needs to specify the source and target mapping that points to the different concepts involved. Each connector has a feature that expresses the meaning of the relation in the semantics. Whenever an instance of the relation can be created, the visualization updates to render the connector. For example, the *mother* relation path automatically changes whenever the end-user changes the position of its source or target. Style can be added to an edge to custom visual properties (e.g., change the color, add a decorator).

2.2.2. Layout management

Graphical editors usually require management of the general disposition of the elements for better visualization. Adapting the position and size of the different elements is a time-consuming activity when the end-user should be focused on building and modifying the model [**29**]. However, most graphical languages, like Visual Paradigm[2] and AToMPM, require user interactions to optimize the representation.

Graph drawing techniques focus on the creation of easily readable graphs that maximizes specific visual aesthetics [**30**]. Visual aesthetics are measurable properties that reflect the quality of a graph (e.g., limiting edge-crossing and edge-bending, making the ability of the end-user to create a mental map easier). Numerous approaches can be adopted when generating the drawing. Layered techniques assign each vertex to layers before rendering the edges [**31**]. Orthogonal drawings represent the graph as a grid with nodes, the edges being either vertical or horizontal [**32**]. Force-directed methods compare the graph to a physical model with actions of attraction and repulsion to place the vertices [**33**]. All of these techniques have their pros and cons depending on the type of graph they generate and the desired visual aesthetics. When considering their usage in an editing environment, an important concern is their ability to support the addition or suppression of a model element without making important changes in the general disposition of the graph.

Initiatives like KIELER [**34**] focus on the integration of automatic layout in graphical modeling tools. As the management of model elements may differ depending on the context, they try to offer interfaces to configure and customize different layout algorithms. The end-user can choose to generate a layout at any time. The selection of different algorithms and options is defined as a *meta-layout*. The pragmatics established in [**35**] strongly encourage the use of automatic layout in graphical representations. They encourage

[2]https://www.visual-paradigm.com/

structure-based editing where the end-user only makes structural modifications to the model. The graphical view is only updated after these interactions. These pragmatics have been applied in [**36**] to create and define statecharts [**2**]. The approach uses two editing methods: a macro-based technique that takes place in the graphical representations and a text-based technique relying on an alternative view of the model. A study performed on the tool demonstrated positive results regarding the user experience. However, this solution focuses on a single formalism. A proper layout with adapted interactions can vary from one DSL to the other, and more general solutions need to be implemented.

2.3. Editing Style

Language workbenches allow the definition, reuse, and composition of languages and their resulting integrated development environment (IDE) [**6**]. As the user editing activities create and modify models that are related to a specific domain, different techniques can be adopted to verify their syntactical correctness.

2.3.1. Free-form Editing

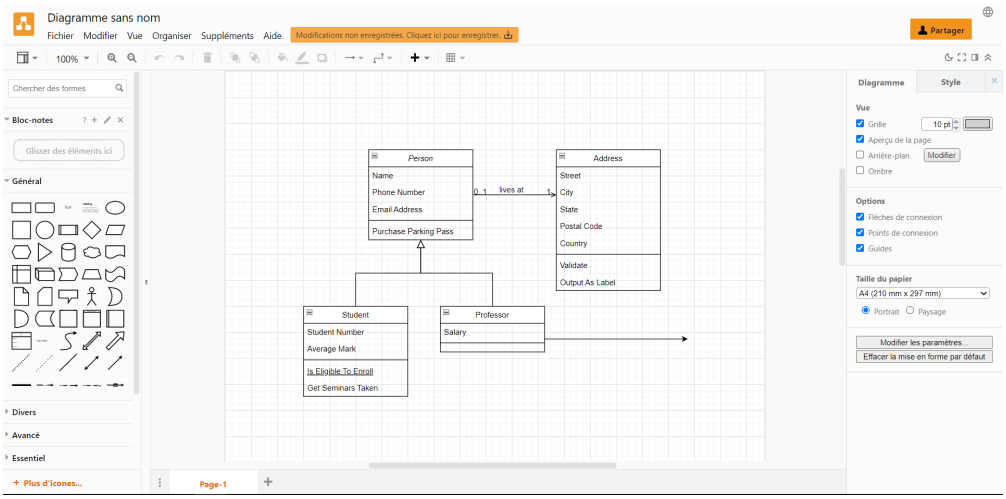


Figure 2.3. Creation of a class diagram in draw.io.

Free-form editing focuses on offering freedom of representation to create diagrams. No analysis is performed to ensure the syntactical correctness of the diagrams. Hence, creating meaningful and well-constructed models strongly relies on the knowledge of the DSL. Figure-2.3 represents a class diagram created on draw.io[3]. In the editor, the user is presented with a canvas and a panel with different shapes that can be drawn. The available graphics are not language-dependent and do not differ from one DSL to the other. The absence of

[3]https://app.diagrams.net/

syntax verification facilitates the addition and modification of graphical figures but makes the presence of syntactical and semantical errors in the model more likely. For example, the diagram shown in Figure 2.3 represents an invalid class diagram because it contains a relation that comes from *Professor* but finds no target. The different diagrams created in the editor are not related to any abstract syntax, thus no verification can be performed.

2.3.2. Syntax-directed Editing

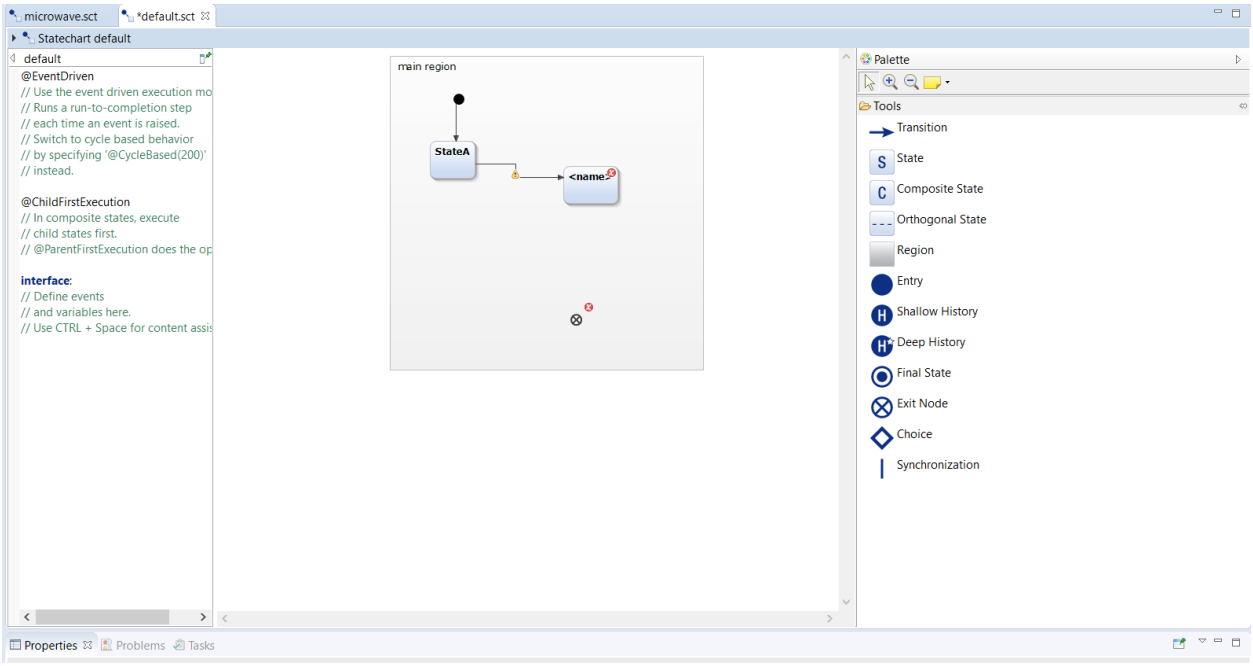


Figure 2.4. An example of a statechart in Yakindu.

Syntax-directed editors use a parser-based approach to generate the abstract syntax graph (ASG) corresponding to a graphical representation. As the end-user performs modifications on the diagrams, the editor analyzes the syntactical correctness of the graph to ensure that no error is present in the model. Validation requires a knowledge of the syntax, so syntax-directed editors are strongly language-dependent. Yakindu [**37**] uses this approach to represent statecharts [**2**]. In the editor shown in Figure 2.4, the end-user is presented with a toolbar to add predefined representations of model elements. This guarantees that the concrete syntax is fixed for the model. When interacting with the graphics, visual cues are added to the canvas to signal violations of the syntax to the end-user. In Figure 2.4 for example, the exit node in the lower area of the main region needs to be connected to a state. The parser-based approach minimizes the presence of errors in the model and helps the end-user to better understand the underlying structure. However, in Yakindu, they require that the she manually corrects them. These different interactions take place in an environment where the end-user already has to manually manage the size

and position of the elements. These properties can only have an impact on the concrete syntax.

In tools like AToMPM, the editing activities are verified by construction to check their validity. The end-user can try to create invalid elements in a model but the editor prevents these operations to be completed.

2.3.3. Projectional Editing

The parser-based approach requires an analysis of the concrete syntax to generate the ASG. To translate the different parts of the representations, constraints need to be established on the different notations used in the language. Projectional editing tries to overcome these constraints by proposing an alternative approach [**9**]. Rather than considering the ASG as a result of the parsing process, the end-user directly interacts with representations of the ASG called projections to guarantee syntactical correctness. Constraints are applied on the concepts, not their projections. For textual DSLs, this approach has allowed the embedding of various notations, such as mathematical formulas or tables, that would otherwise be hard to parse [**11**]. In addition to freedom of representation, projectional editing allows the end-user to focus on the task at hand rather than the concrete syntax. This approach can be promising when considering the need for structure-based editing defined in Section 2.2.2.

2.4. Projectional Editors

The definition of projectional editing can be traced back to the 1980s [**9**]. The Incremental Programming Environment presented in [**38**] focuses on a dedicated environment for compiler-based programming languages. The approach proposes to centralize the different tools necessary to the programmer (the editor, the translator, the linker and loader, and the debugger) in a unique system. A similar process can be found in GANDALF [**39**] and The Synthesizer Generator [**40**]. Modifications of the textual syntax are based on templates with "holes" that the end-user can fill. Modern solutions like the Meta Programming System (MPS) [**10**], Gentleman [**41**], and the Whole Platform [**42**] consist of frameworks for the definition and generation of language-specific projectional editors.

For this thesis, we created an extension of Gentleman for graphical and projectional editors. Some of our choices of implementation were inspired by MPS and the different studies performed on it. In the following subsection, we present these two frameworks.

2.4.1. MPS

MPS has been developed by JetBrains, a Czech software development company. It is a language workbench based on projectional editing. It allows the definition of domain-specific and general purpose programming languages (e.g.,Java, C, C++), and offers support for operations like code generation. MPS has already been used for multiple projects like mbeddr [**43**], a set of languages for embedded software engineering, or a real-time Java development environment [**44**]. The language workbench is composed of various DSLs dedicated to specific aspects of a language [**45**].

Structure. The *Structure* defines the abstract syntax of a language. The definition of metamodels in MPS is similar to object-oriented programming. Concepts can be extended and implement different interfaces. As the internal representation of a language is an AST, the *Structure* is defined with a root concept. Concept attributes are divided into three categories. The *properties* are described with primitives like strings or numbers. *Children* represent relations of composition. They are typed with concepts that are defined in the metamodel. Finally, a concept can have *references* with cardinalities to point to specific instances of a model.

Editor. After defining the metamodel of a language, the second step is to create projections for the concepts. For each concept, the language engineer can define an *Editor* that will be used as a view and a controller. An editor is composed of cells that are organized in a layout. Each cell can be typed relatively to the concept to represent children, properties, references, or static components. A concept can be represented by no more than one editor. As each editor works as a controller, actions can be attached to different cells and keys. Style properties can be defined to offer a larger diversity in the notations.

Generator and TextGen. MPS offers support for model transformations. In the *Generator*, templates and rules can be created to define model-to-model transformations. Templates use the output language to write the results of the transformation using its cell editor. The mapping between a concept of the input language and the templates of the output language is described in the rules. In addition to model-to-model transformations, MPS allows for code generation with the *TextGen*. It is composed of multiple operations that are applied to the concepts to print text in a designated layout configuration.

2.4.2. Gentleman

Gentleman is a lightweight and web-based projectional editor generator developed at Université de Montréal. Since most solutions are heavily platform-specific, Gentleman uses web technologies to create projectional editors and model instances. In an experiment conducted in [**46**], Gentleman has demonstrated a usability and understandability that surpassed MPS for modeling activities. The editor can be embedded in various systems, the thesis demonstrating an integration in ReLis [**47**], a tool that focuses on systematic reviews. To explain the different structures that define an editor in Gentleman, we will use the example of the *MindMap* editor available as a demo on the website of the editor.[4]

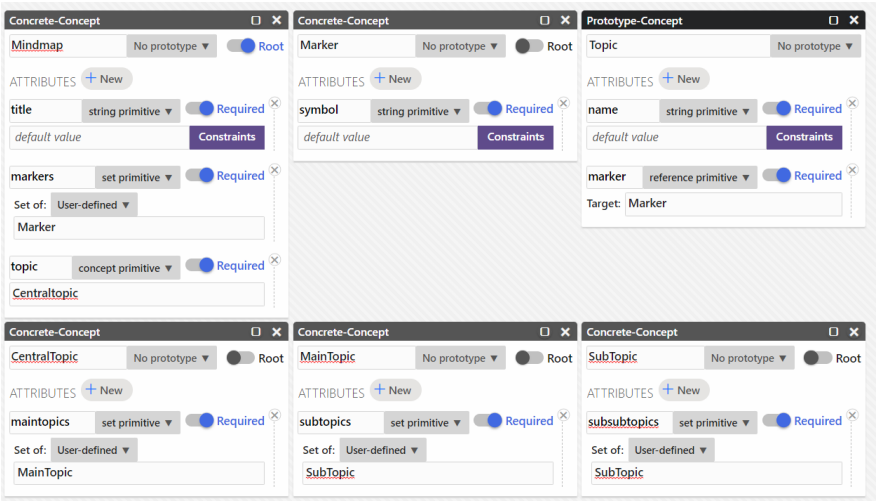


Figure 2.5. Concept definition of the Mindmap on Gentleman.

Concept Model. To generate editors with Gentleman, the language engineer has to create a model defining the concepts and another model for their projections. The concept model describes the metamodel of a language. It contains primitive concepts (string, number, boolean, reference, set) and user-defined concepts. Attributes are identified by a name and typed by a concept. Inheritance and extension are respectively realized with prototype and derivative concepts. Figure 2.5 shows the definition of the metamodel of a *Mindmap* editor. The root concept is the *Mindmap*, with a title defined as a string, a set of *Marker*, and a *CentralTopic*. Markers are decorated with a string and can be referenced by the various *Topics* in the model. The *Topic* prototype is inherited by three concepts: *CentralTopic*, *MainTopic* and *SubTopic*. Their respective attributes allow the creation of a hierarchical structure where a *CentralTopic* can contain *MainTopics* that are composed of *SubTopics*.

[4]https://geodes.iro.umontreal.ca/gentleman/demo/mindmap/index.html

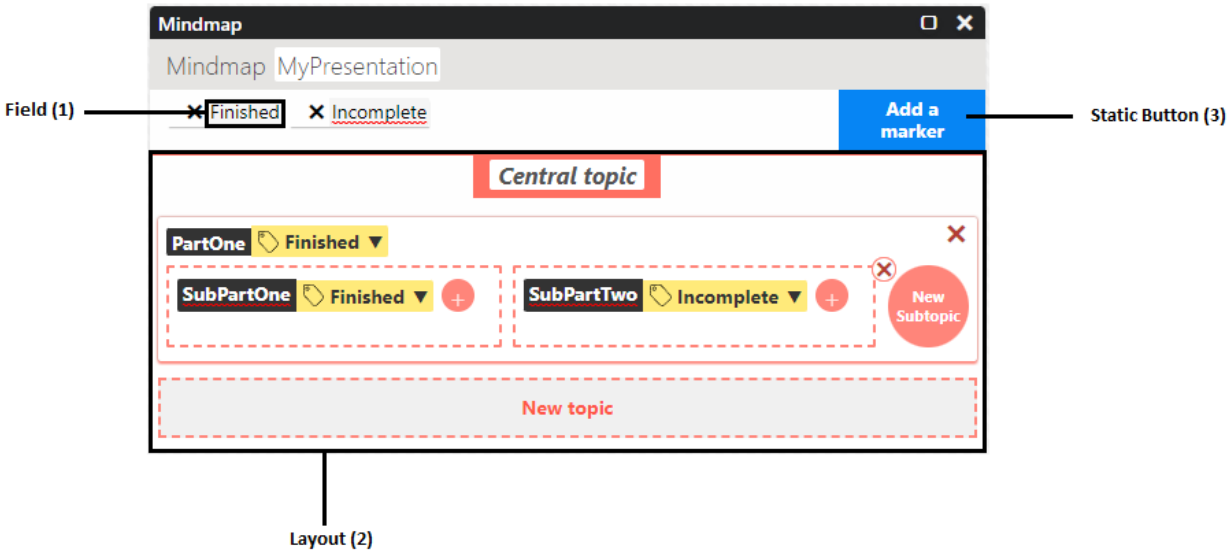


Figure 2.6. The MindMap projection.

Projections. The projection model groups the projections of the different concepts found in the concept model. A projection is mapped to one concept. Gentleman uses *container-based* projections using multiple layers to create a representation. *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, end-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.

Figure 2.6 represents the projection for the *MindMap* concept. The highlighted *text-field* (1) allows setting the symbol of a marker. The *layout* (2) represents a CentralTopic. It organizes the placement of three elements: a static-text with the name of the concept, a *list-field* to represent the set of *MainTopics*, and finally a *static-button* to add new items to the set similar to the one shown in (3). The layout uses a flex disposition with a vertical flow.

Export and Integration. After creating the different models representing an editor, concepts and projections can be built to generate a JSON file describing their structure. Generating an editor requires loading both models. A configuration can be added to

customize the toolbar. The resulting editor allows the instantiation of concepts and attributes to create new models. Models can be exported in a JSON file or loaded in the editor to pursue editing activities. Gentleman can easily be integrated into web pages by decorating an HTML tag or loading an editor with Javascript [**41**].

2.5. Motivations

This chapter demonstrated the advantages of creating languages tailored to a specific problem domain. As a model needs to be syntactically correct to be reused, multiple editing strategies can be adopted with different degrees of freedom for the end-user. Projectional editing offers an interesting paradigm to help the creation of meaningful and well-constructed models. However, current projectional editors focus solely on textual notations, even though a graphical concrete syntax could be more suited to specific DSLs.

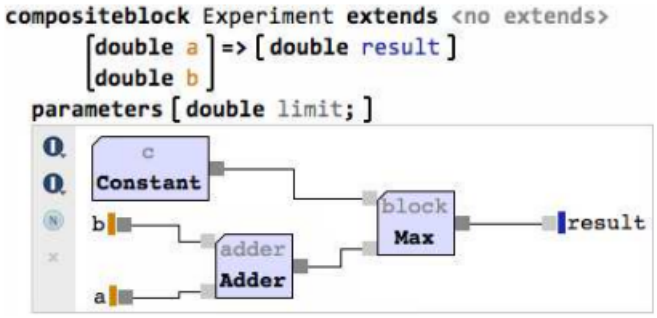


Figure 2.7. A graphical editor using MPS.

Currently, MPS offers no support for projectional graphical notations. In [**10**], the authors describe a graph-like representation that can be embedded in a regular MPS editor. Graphical projections are declared as diagrams with blocks that can be connected using ports visible in Figure 2.7. The paper describes these notations as "not yet as mature as the rest of MPS" and their creation as "not yet as convenient as it should be". MPS documentation has been updated since to signal that this functionality was deprecated.[5] In MBEDDR, a programming language and IDE for software engineering based on MPS, the end-user can create and interact with diagrams using a language developed with teams from MPS. They explain on their website that Jetbrains is currently working on a framework to define graphical editors.[6]

[5]https://www.jetbrains.com/help/mps/diagramming-editor.html
[6]http://mbeddr.com/2014/11/14/graphical.html

In Gentleman, graphical notations can be added to a projection as static decorators. Images can be loaded from the web using a URL. The general structure of Gentleman relies strongly on container-based projections. Layouts use CSS to create tabular forms or define a general flow of elements disposition. Static projections are used to adapt the organization of the textual components.

In this thesis, we propose the definition of a new type of editor that is graphical and projectional. As a demonstration, we extended Gentleman to generate graphical editors.

Chapter 3

Projectional and Graphical

"Projectional editors are editors where a user's editing actions directly change the abstract syntax tree without using a parser." [**9**]. The essence of projectional editing relies on interaction. When the end-user performs an action, the resulting effect must be tailored to the semantics of a language. However, the interaction might seem unusual for non-expert users [**48**]. The new set of operations available with graphical concrete syntax (moving objects, dimensions management, modifications of the path of an edge) needs to be examined when considering a projectional editor. We created an extension of Gentleman that focuses solely on graphical projections. The definition of new structures for interaction and visualization of the model allowed us to define new constraints that are inherent to graphical and projectional editing.

3.1. Extension of Gentleman

Gentleman already has a structure to define, modify, and export concepts and container-based projections [**41**]. The main guideline of the extension was to focus on addition of new projections and components without modifying the existing system.

3.1.1. Representing graphics

Because Gentleman is a web-based editor, two options were available for creating graphical projection: raster-based or vector-based elements.

Raster Images. Raster images consist of a grid with a description of its pixels [**49**]. Multiple formats can be used for web applications such as JPEG or GIF for example. When it comes to making the graphics interactive, the HTML5 Canvas element is the way to go. Canvas uses JavaScript to describe each step of the drawing and additional methods. Each pixel is then created when the figure is rendered. On the one hand, Canvas is really precise because each pixel is drawn individually. This can be a good asset for systems with a high

level of detail. In addition, a lot of animations are available by default, making the graphics more dynamic. On the other hand, the pixel-based approach implies that each image has to be redrawn when modifications occur. This becomes less efficient for layouts with a lot of information where the computation of coordinates can be impacted by the presence of other elements. Another concern is that Canvas elements do not track the state of the shapes they contain. Thus, the mapping between the ASG and its projections is harder to maintain.

Vector Images. Vector images are constructed by creating shapes while specifying attributes such as coordinates and size in a container. Scalable Vectors Graphics (SVG) is a language describing two-dimensional drawings using XML [**50**]. Vectors can be styled using CSS and interacted with by defining scripts. One key aspect of SVG is that it is represented in the DOM when rendered. Modifications are directly applied to the targeted components, without the need for a redrawing of the complete image. As it is vector-based, SVG has some additional pros and cons. On the one hand, it is well suited when it comes to scalability. Vector-based images are resolution-independent because coordinates and length are relative to the viewport. On the other hand, the rendering of images containing a lot of elements may not be cost-efficient, because each vector has to be drawn. This last point makes sense when it comes to displaying large models, but the advantages of staying in the editing world make it not so important. Indeed, editing activities often focus on parts of the model, and as elements are only rendered once, this cost limitation only has an effect during the loading phase of large models. Moreover, SVG as already proven its utility as a concrete syntax [**51**]. Because of these reasons, we chose to use SVG to describe graphical projections.

3.1.2. The projection model

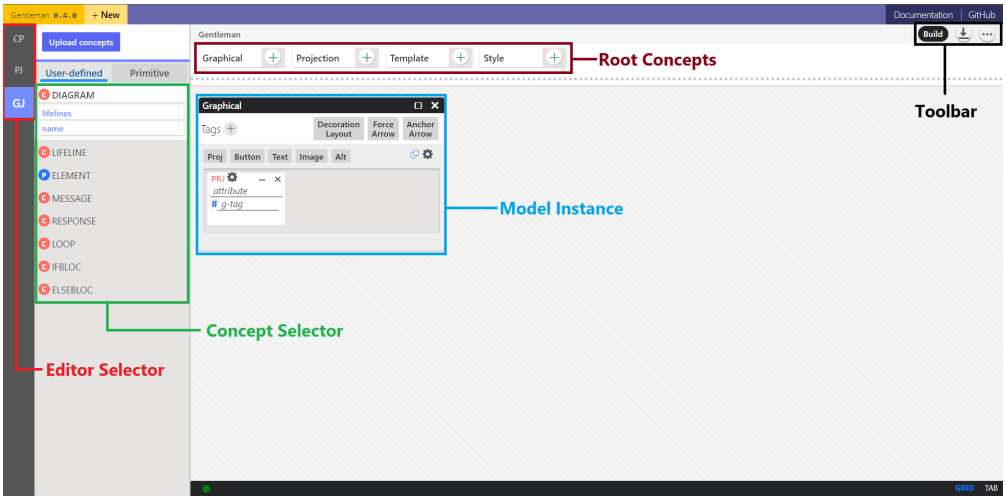


Figure 3.1. Gentleman graphical editor.

When the language engineer first uses Gentleman, he is presented with three different editors: the concept editor (metamodel definition), the projection editor (container-based projections), and the graphical editor. As Gentleman is bootstrapped, each editor is defined with its own set of concepts and projections. The graphical editor is shown in Figure 3.1. Interacting with these editors uses the same process. Concepts can be added to the main area and directly modified by the language engineer. When the model is ready, the build button starts the export. In the graphical editor, some projections are strongly related to some concepts. For example, an edge with an origin and a target might not be very relevant to represent the name of a person. After selecting a concept, the language engineer is presented with a restricted set of available projections to prevent any incompatible representation.

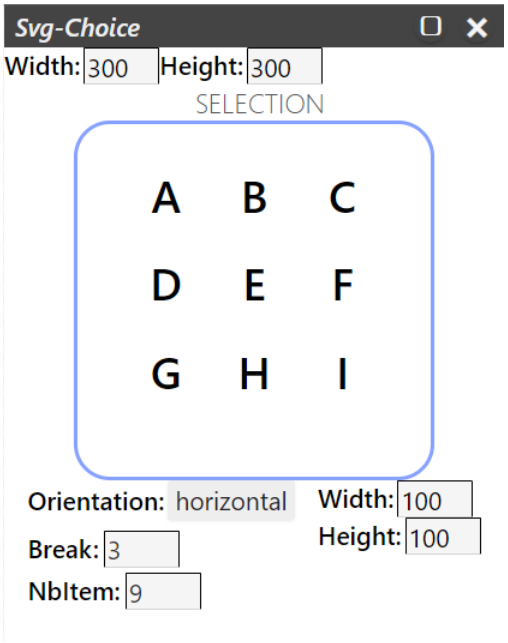


Figure 3.2. A simulation for a *Choice-Field*.

Overview. The graphical model uses the same structures as the container-based projections. *Fields* are used to define attributes or primitives, and to interact with them. *Layouts* organize the elements. Because different dispositions may be desired, each layout comes with specific parameters and coordinates management. Finally, *Static* elements can be used as buttons or decorations for a projection. For structures that are specific to graphical representations, we added two new types of projections: *Edges* and *Simulations*. *Edges* are used to create connectors. Because edge management might be impacted by the concept or the general disposition of the projections, each layout is related to a specific type of edge so that it can easily be manipulated. *Simulations* are only available in the graphical model,

they cannot be exported by the language engineer. When creating graphical structures, some information such as size or position might need some consideration to prevent visual issues. As the language engineer modifies a graphical projection, a simulation allows her to visualize in real-time the effects of her changes. In Figure 3.2 for example, the layout engineer is presented with a simulation that reflects the effect of the different parameters in a *Choice-Field*. She can modify the orientation of the selection or the dimensions of the choices, and directly see the impact on the resulting projection.

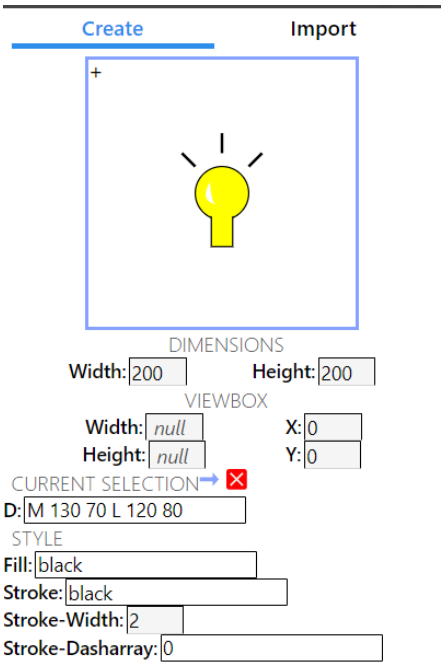


Figure 3.3. The Shape Editor

Creating the graphics. Graphical projections use SVG to create graphics. For export purposes, information on the drawings is stored as a string. Generating SVG with a textual notation might lead to difficulties in global visualization. Moreover, creating an SVG file, going to the source code, and pasting it into a text area on Gentleman might be a time-consuming activity for the language engineer. The *Shape* editor, shown in Figure 3.3, allows the creation of SVG graphics directly in Gentleman. The root concept of its metamodel is the *Canvas*. It can contains multiple elements declared as *Shapes* and spread into *Groups*. Each concept is related to its corresponding SVG vector and they share similar attributes. The editor is composed of two different views: a graphical projection, for visualization and basic editing, and a container-based projection to perform very specific operations on the vectors (e.g., modify the vertices of a polygon). Existing SVG files can still be reused in Gentleman by importing them [**52**]. The editor parses their content and

generates the corresponding SVG to show it to the language engineer.

Using the container-based projections. Having textual notations embedded in a graphical editor can be a real asset for productivity [**26**]. The editor renders each container-based projection in a separate `<div>` and automatically organizes them with CSS rules. The structural aspect of these elements creates some constraints on their size that are hard to manage for an SVG element. The computation of `<div>` elements dimensions requires information on their parent element. In opposition, adding HTML into SVG demands that its width and height are known and fixed. To prevent any issue, the embedding of container-based elements in graphical projections is not available in Gentleman. Instead, we created alternative structures using SVG. 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.

Export and import. When the projection model is complete, the language engineer can export it in a JSON file. Building container-based or graphical projections uses the same procedure, the only difference being that the analyzed structures are not the same. To generate the editor, the user only has to import the JSON files for concepts and projections. Once the end-user interacts with Gentleman, the editor makes no distinction between graphical or container-based projections.

3.2. Interaction-oriented projections

Defining interactions in a graphical environment can be challenging [**53**], especially when the interaction can be specific to the context of the resulting editor. Moreover, a study performed on MPS [**9**] supports the idea that the editing activities in projectional editing still need to be improved to maximize the efficiency of the user. The following subsections describe the different actions to perform on the ASG and how they can use the new information available with graphical languages.

3.2.1. Creating and deleting concepts

Adding new concepts or removing them is a basic mechanism to make the ASG evolve. Pressing keys (e.g., Enter) is considered the default user interaction to add concepts or static representation to a textual language. Considering the end-user as a clear visual representation of his position in the AST, the effect is easily predictable. For the ASG, the positioning can be more difficult to read, as graphical projections can contain multiple layers and shapes. If multiple projections are centralized in a restricted area, clicking on a specific one and then pressing a key might be harder than it needs to be, especially for

relatively small targets [**54**]. An alternative solution is to rely on graphical projections that are represented as buttons. If a projection can only be related to a single concept, a concept may have multiple projections. The idea would then be to use the layout solely to interact with the elements it contains, and delegate creation and suppression to buttons that can be separated from areas that are crowded with information. The disposition of the buttons can then be decided by the language engineer to guarantee the ability of the end-user to predict their effect [**55**].

Implementation. In Gentleman, the language engineer can directly create static buttons using SVG and bind actions to them. Because actions are very context-related and the graphical model is only an extension of the container-based model, the only actions available are "CREATE", "CREATE-TREE", "OPEN-SIDE", and "DELETE". When a concept is optional, the language engineer can attach the delete button to the projection with specific coordinates and dimensions. For prototypes that can have multiple values (hence multiple potential projections), a mapping between the value of the concept and the coordinates of the button can be created.

3.2.2. Select a value

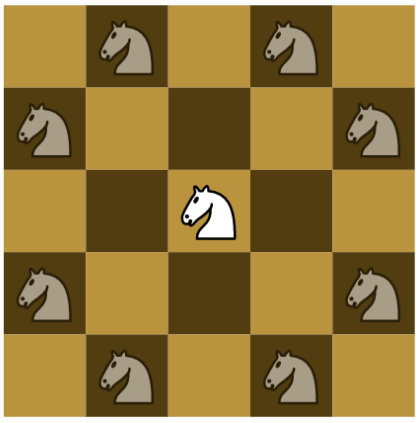
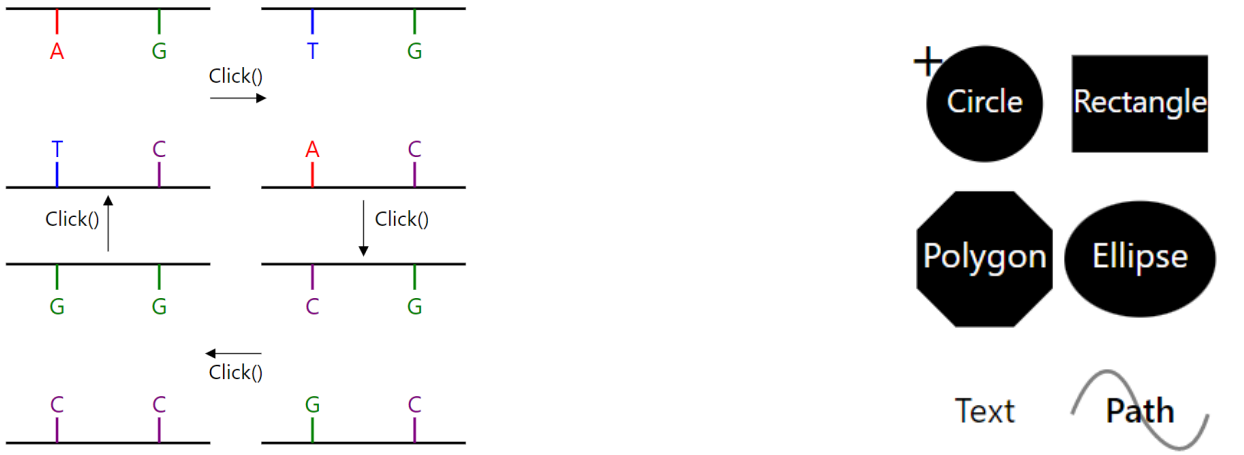


Figure 3.4. Using coordinates to display the possible moves for a knight.

Constraints can be applied to the possible values of a concept. For example, when creating a reference, the end-user has to decide on its target. Textual and container-based projectional editors such as MPS [**10**] rely strongly on auto-completion to ensure a good

understanding of the possibilities and syntactical correctness. Displaying choices in graphical languages can be realized similarly, and even extended using the additional visual information that can be communicated to the end-user. Concepts can have an impact on the properties of a projection (e.g., dimensions, coordinates, shape). For example, the representation of the concept named *RedCircleWithYellowStroke* might offer some suggestions on projections that might be related to it. Exploiting this information might be a good way of better communicating to the end-user the possible choices he is offered.

Figure 3.4 represents a knight on a chess board. When moving the piece, only specific squares can be reached. Rather than showing the coordinates in a textual list next to the knight, the editor directly displays the possible results of the movement. To select the next position, the user only has to click on the one he wants to reach. In this example, the possible values for the move are coordinates. Using these coordinates enforces the end-user visualization of the choice and limits the presence of interaction boxes that could potentially overlap other projections (e.g., another piece that is in the neighborhood of the knight.). Hence, graphical projections can (and should) take advantage of the visual information contained in the semantics to help and diversify the user interactions.



(a) DNA editor with a *switch-field*

(b) Shape selector with a *choice-field*

Figure 3.5. Examples of the *switch-field* and *choice-field*.

Implementation. Three graphical projections were created in Gentleman to select a value: the *choice-field*, the *placeholder-field* and the *switch field*. The choice-field is an SVG element that displays the potential values for a concept in a tabular form. The language engineer may specify the order and the dimensions of the available choices. Each choice

has a specific projection for the choice-field that is identified with a dedicated tag. The placeholder-field uses the relation between the coordinates and the concept to show the potential values. Finally, the switch-field rotates through the possible values of the concept whenever the user clicks on it. The order of rotation can be defined by the language engineer.

Example. Figure 3.5 shows examples of these fields in different editors. On the left part, we have an editor that allows us to create DNA sequences. The base found in the sequence can have four types: adenine (A), thymine (T), guanine (G) or cytosine (C). Each base has specific match on the other strand, A with T and G with C. By clicking on the switch-field, the value of the pair changes by rotating in a specific order. The right part of the figure can be found in the *ShapeEditor*. When a new shape is added to the model, a choice-field displays the available vectors. Clicking on a vector sets the value of its prototype. Finally, Figure 3.4 is an example of a placeholder-field.

3.2.3. Textual projections

Concepts with an unbounded range of values (e.g., String, Number) are strongly related to textual representations. In an environment where these data can evolve and be modified by the user, creating only a graphical representation would not make much sense because of the unpredictable values of the concept. When constraints are applied on a String (e.g., there are only twelve values to describe a month of the year) a mapping can translate values to graphical elements or properties.

Figure 3.6. The text-field in Gentleman.

Example. The *text-field* can simulate the interaction that is expected by the user when typing text, as seen in Figure 3.6. When the user clicks on it, the position is translated into SVG coordinates to analyze the closest character. After resolving the position of the click, an SVG rectangle is created to simulate a cursor, and each key typed is controlled by the field and its concept to validate the modification of its value (e.g., no letter for an integer.). A text-field comes with an anchor to set its position and a placeholder to ensure that it is always visible, even if there is no value. In addition, the language engineer can add style

using CSS. This projection simulates a text area for textual input, a central interaction in an editor [**56**].

3.2.4. Managing the visualization

Visual aesthetics play a crucial role in the ability to communicate information to the user [**57**]. Creating multiple interaction-oriented projections may hinder the ability to fully understand a model, especially if they are crowded in a very restricted area. Moreover, modifications of a specific part of the ASG may be very context-dependent (e.g., if the user is working on concept B, having boxes for selection or an alternative view of concept A may not be very useful). Hence, interaction-oriented projections should be obvious enough for the user to understand the actions he can perform, but not obstruct information on the underlying concept once the editing has ended.

Example. In Gentleman, choice-fields, placeholder-fields, and any type of alternative visualization can be closed or opened by the user when needed. These actions are performed using buttons that can be directly placed by the language engineer.

3.2.5. Metamodel constraints

In projectional editing, a model must always be syntactically correct. The result of an interaction with a projection must respect the different constraints that can be applied to a concept. Since projections are representing the model, these constraints have no impact on their management. Indeed, projections can only represent syntactically valid elements. In Gentleman, the concept analyzes the result of an interaction. If it is not considered correct, the concept does not accept the result and stays unchanged. The projection then adapts itself to ensure consistency. Since metamodel constraints are not handled by the projections, we did not create any additional structure to manage them.

3.3. Inherited requirements

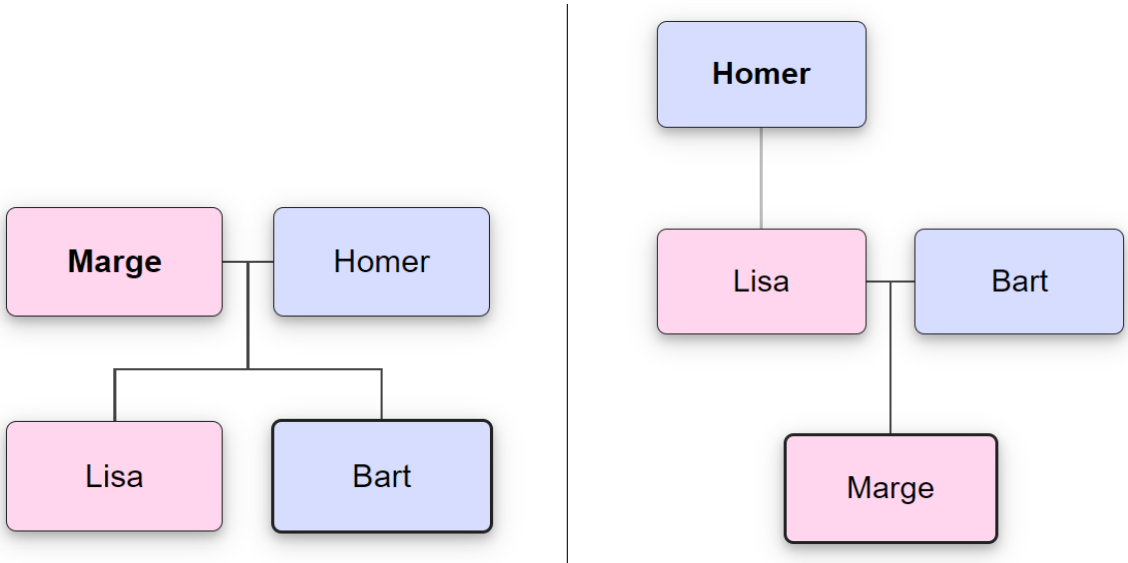
The previous point focused on defining the interaction with graphical projections. From the different structures we implemented in Gentleman, we extracted basic requirements that are inherent to these interactions. They are described in the following subsections.

3.3.1. Visualization vs. Projectional

Before going into the definitions, it is important to make the distinction between graphical projectional editors and visualization tools, such as UMlet [**58**]. They focus on representing model information. Usually, the editing activities take place in a textual environment represented, for example, as a side window. As the user performs actions on the textual model,

the graphical representation adapts itself to guarantee consistency. The model transfers the changes to the graphics. If this mechanism also exists in projectional editing, the relation between concepts and their representation is bidirectional. Interacting with the projections is the designed mechanism to modify the ASG, so the interaction must take place in the graphics. For visualization tools, this is not a mandatory requirement. Even if projections may only represent parts of their related concept, ***the focus should be to keep the interaction in the projections, hence, in the graphics*** **(Req #1)**.

3.3.2. Automatic positioning



(a) Marge is a mom

(b) Marge is a daughter

Figure 3.7. Marge's family tree.

In graphical languages, the layout disposition plays a critical role. Positions of the projections might be dictated by their underlying concept. Let us take for example Figure 3.7 representing a family tree (a structure to help visualize genealogy). In this model, each member of the family is represented by a rectangle containing its name. The color indicates the gender. For a dedicated person, all of its ancestors are contained above in the tree, and all of its descendants are displayed below. By comparing the left and the right models, we see different information. In (a), Marge is the mother of Lisa and Bart, whereas in (b), she is their daughter. The difference between the two models is explicitly displayed by the position of Marge in the tree. To create such structures in non-projectional editors such as AToMPM [**8**] or Sirius [**27**], the end-user would add the concepts to the canvas, place them, and then connect them to the corresponding elements. This process creates an issue with the ASG because it starts by displaying an element that is not connected to anything, which does

not make sense in the context of a family tree (unless it represents a separate family tree). Since in projectional editing, projections are direct representations of the ASG, this type of syntactical error is not allowed. Elements can only be displayed when the ASG is correct, not before. That is why ***the positioning of the object must be directly managed by the editor*** **(Req #2)**. The constraint of automatic positioning might be delegated to a layout representing a collection of elements or directly to the concept if coordinates are a relevant attribute in the abstract syntax.

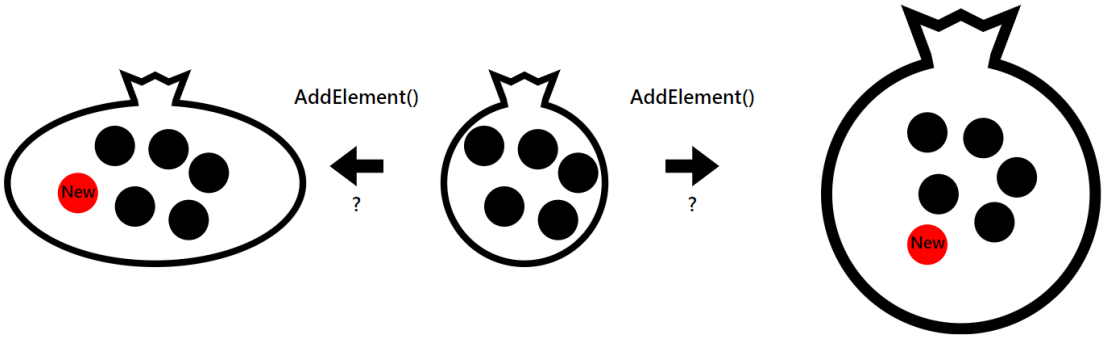


Figure 3.8. Possible evolution of a layout for a set of elements.

3.3.3. Automatic sizing

Adding or removing concepts results in changes in the amount of information displayed in a projection. As concepts may have unlimited possible values, these operations can have an impact on the dimensions of the layout that contains their projections. For textual languages, the general organization of the language can be compared to a tabular organization. Creations of new projections only result in the evolution of the horizontal/vertical flow, using spaces, line breaks, or indentations. Since graphical languages offer more axes for the transformations of the components, managing the dimensions modifications is a challenge when considering a projectional editor.

In Figure 3.8, a set of elements is represented by a bag. When the bag is full and a new element is added, the language engineer can decide to make the bag wider, as seen on the left part. Another might decide to increase the width and height of the bag as seen on the right part. The choice between the two evolutions only has an impact on the concrete syntax but is essential to have a clear visualization of the items of the collection. In non-projectional editors like AToMPM [**8**], such transformations usually require that the end-user manually modifies the dimensions (e.g., click on the corner of the graphics and drag until the desired width and height are met). Since these interactions have no impact on the ASG, they are not relevant for projectional editing. Hence, ***the editor must be***

able to automatically compute dimension modifications **(Req #3)**.

The first step is to define the evolution of the projections during additions and deletions of concepts. This can be directly done by the language engineer. The second step is to identify the projection that should manage this transformation. Increasing or reducing the size of a graphic that is located in an area that is crowded with projections may lead to overlapping issues. As a layout already has information on coordinates, it is best suited to manage the size of its elements. After resolving the changes, it should be able to communicate it to other projections that can be impacted by the new dimensions. This type of communication implies that the projections that the affected projections must be easy to identify.

3.3.4. Interacting with the projections

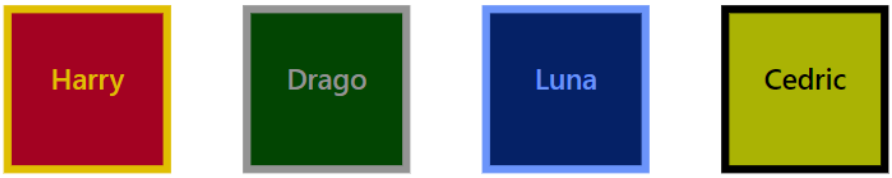


Figure 3.9. Four players in different teams.

In a textual or container-based editor, the structural aspect of the language facilitates an easy approach to the interaction. Using checkboxes, text areas to fill, or auto-completion gives hints to the end-user on the operation he can perform on the ASG and their potential results. For graphical projections, the identification of these interaction points might become more difficult. Graphics are composed of multiple shapes (e.g., ellipses, polygons, curves) that can overlap, be disposed in different layers, or use different colors. Modifying the ASG may result in numerous changes for different parts of projections but more specifically, a single projection can be impacted by multiple attributes. Figure 3.9 shows four different instances of the concept *Player* from a soccer league. Each *Player* has a name displayed in the center of the projection, and a *Team* that can be interpreted from the colors of the graphics. Changing the team affects the background but also the color of the name. The end-user can modify the name by clicking on its projection and typing characters. Here, the text is impacted by the two attributes (name and team) of a player which may lead to confusion. To ensure a clear understanding of the available actions, ***each interaction point must be designed with its own projection*** **(Req #4)**. Dividing interaction components allows the end-user to have a clear view of the modifications he can make to the abstract syntax. When the resulting operations are completed, the concrete syntax

adapts itself to the new state of the ASG. The effects of these changes can be applied to multiple projection properties (e.g., color, size, borders). Identifying the targets of these modifications can be facilitated by creating a mapping between the new value of a concept and the effects it has on different sets of graphical components.

3.3.5. Visual aesthetics

A key aspect for the end-user is to have a global understanding of the projections he is interacting with. The process of decoding the information contained in a model can be facilitated by taking the general organization of its representation into consideration [**59**]. As size and coordinates cannot be adjusted manually, a layout must also consider visual aesthetics to guarantee a better representation of the underlying model [**60**]. Managing edge-crossing, avoiding overlapping, or offering a mental map are key elements that make not only a model easier to read [**30**], but also more pleasant and simple to use [**61**]. Because of the need for easy identification of the interaction points, focusing on these aspects also guarantees that these projections are reachable. Hence, ***visual aesthetics must be considered when designing a layout*** **(Req #5)**. This requirement supports the idea of *meta layout* discussed in Section 2.2.2.

Chapter 4

Connectors and edges

Drawing graphs requires the consideration of two components: nodes and links. As concepts can reference each other, creating edge projections becomes an important requirement for a graphical and projectional editor. In this chapter, we only consider edges and connectors with one origin and one target.

4.1. Concept analysis

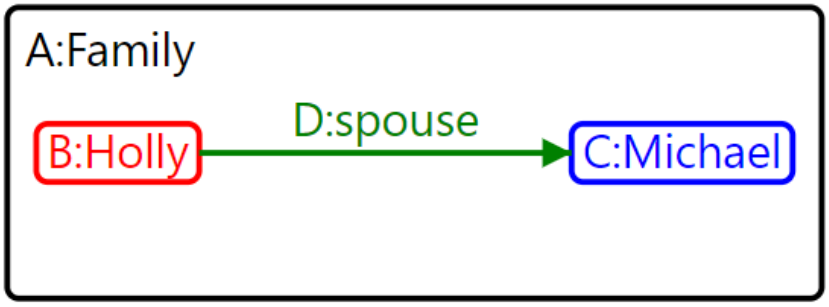


Figure 4.1. Example of an edge in a graphical representation.

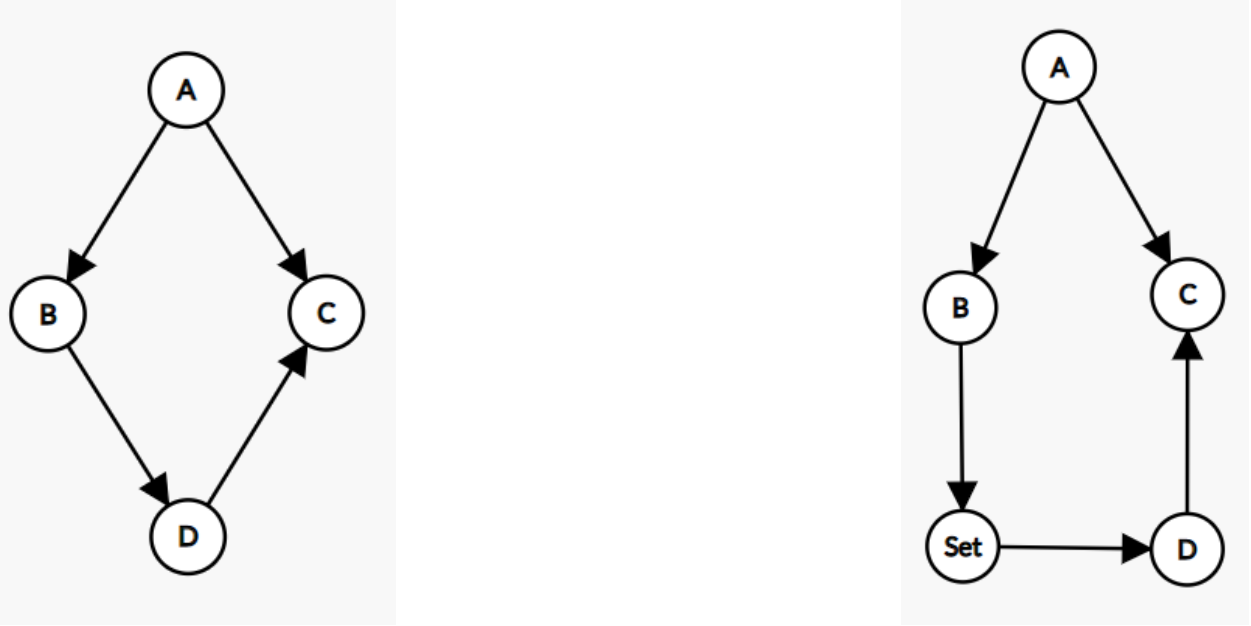
Projections are representations of concepts. Before working on the creation of connectors, it is important to understand the kind of concepts they can represent. Edges connect two elements by displaying a line that can carry additional semantical information (e.g., extensions and implementations relations are rendered differently in a UML class diagram). Figure 4.1 illustrates the different concepts we use in this section. A person named *Holly* (B) is married to *Michael* (C). The two persons are represented by rectangles that are displayed in a layout corresponding to the concept of *Family* (A). The relation is represented by the *spouse* (D) concept. Depending on the position of this concept in the ASG, we get different

requirements for its projection definition. The different concept definitions in the following subsection will be based on a structure similar to the one found in Gentleman:

- A concept can be defined with a name and a set of attributes or a built-in primitive.
- A type can be a primitive or a user-defined concept of the model. Effectively, a type refers to a concept.
- An attribute has a name and a type.
- A set accepts a specific type.
- A reference targets a specific type.

For example, in Figure 4.1, the concept B can be described as a person with a string attribute representing its name. Depending on the metamodel, the definition of the projection for concept D may differ in order to find the two projections that should be connected. In the following subsections, we explore three scenarios and provide solutions to render the edge.

4.1.1. D is an attribute of B or C



(a) B has a reference to C.

(b) B has a *Set* of references with one pointing to C.

Figure 4.2. Two potential positions of D in the ASG.

The first case scenario is that B has an attribute typed as a reference to another concept, as seen in Figure 4.2(a). For example, Holly (B) can be married to Michael (C).

```
1 B {
```

```
2     name: "Holly",
3     spouse: { name: "Michael" }
4 }
5 C {
6     name: "Michael",
7     spouse: { name: "Holly" }
8 }
```

The connection is created when the value of the *spouse* attribute is defined. On the projection side, this scenario has two main implications. First, the edge projection needs to find its target projection. As multiple projections of C may exist in the editor, finding the right one requires having a structure for identification. In Gentleman, each projection of a concept is identified with tags so that the editor knows which one to render in a specific context. The next point is to find the projection of B. B can be the direct parent of D, just like in our example. Another option might be that B has a set of references as an attribute, as seen in 4.2(b). For example, a *Person* can have multiple *children*.

```
1 B {
2     name: "Holly",
3     children: [ { name: "Michael" } ]
4 }
5 C {
6     name: "Michael"
7 }
```

Each reference is corresponding to an instance of the concept D. In this case, the parent concept is a *Set*. As B is not directly related to D, the edge needs to look higher in the ancestors of the reference. As concepts can become more complex, finding the source of the connector might differ from one DSL to another. One solution is to apply a principle that can be found in MPS. When the language engineer creates a reference, she can add a scope to limit the eligible concepts to specific parts of the ASG. In our case, we can define the scope as all elements between D and its source concept. Defining D as an attribute of C has similar implications.

4.1.2. A contains a connection from B to C

In the second scenario, the concept A has a *Set* of D as an attribute, as shown in Figure-4.3. We can consider for example that the Family is defined with a *Set* of *relations*.

```
1 concept Family {
2     attributes: [
3         { name: "relations", type: Set, accept: Relation}
```

```
4     ]
5 }
6
7 concept Relation {
8     attributes:[
9         { name : "from", type: Reference, accept: Person },
10        { name : "to", type: Reference, accept: Person }
11    ]
12 }
```

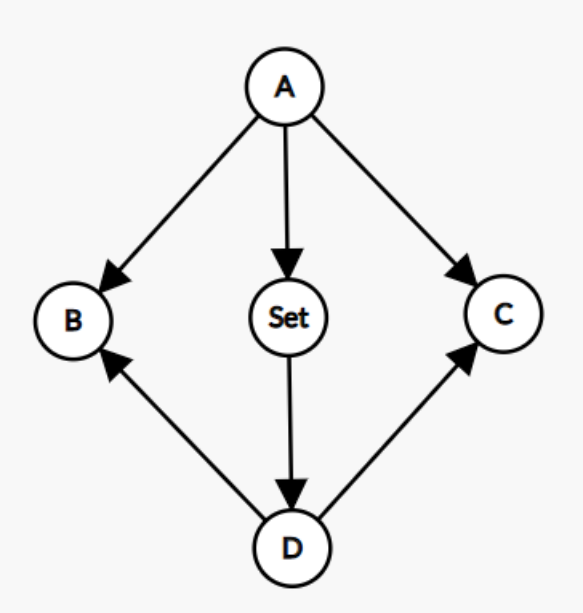


Figure 4.3. A contains a set of D.

As A is already represented by the layout containing the projections of B and C, finding the projections of the source and target becomes easier. When the edge is rendered, it communicates with the layout that has information on coordinates (**Req #2**). The identification can be performed easily, by storing the mapping between a projection and its underlying concept in the layout.

4.1.3. D has no relation with A, B, or C

The last scenario is that D has no relation with A, B, or C, but is still part of the ASG. D is defined as a concept that is not contained in A but has a source and a target that are represented by B and C respectively. Finding information on the projection coordinates becomes more complex because the edge needs to find the right projections of B and C and then find the concept A represented by the layout that contains them, to limit the possibility of edge-crossing for example (**Req #5**). In Gentleman, the identification of projections uses

a descendant mechanism, because it is container-based. When the projection for a concept needs to contain a representation of its attributes, it uses a *tag* to find it. Since D has no direct relation of composition with another concept, this process cannot be used here. The solution we implemented is to extend the concept of tags for graphical layouts. Each layout comes with an attribute called *r-tag*, to identify it as a receiver. When the language engineer defines an edge projection, she gives it information on the *r-tag* of the layout that will contain projections for the potential sources and targets. The connector then looks for the layouts with the corresponding *r-tag* and to find the projections when it is ready to be rendered. Rather than implementing each solution proposed in this section in Gentleman, we manage all scenarios by using *r-tags*.

4.2. Interaction definition

The rendering of the edges is a direct result of end-user interaction. In the graphical state machine editor created by mbeddr [**62**], the creation of transitions relies on click-and-drag. Even if this editor cannot be considered as projectional according to our requirements (e.g., states can be moved or resized by the end-user), this interaction could be interesting to diversify the available operations and keep them in the graphical projections (**Req #1**).

4.2.1. Defining the click-and-drag

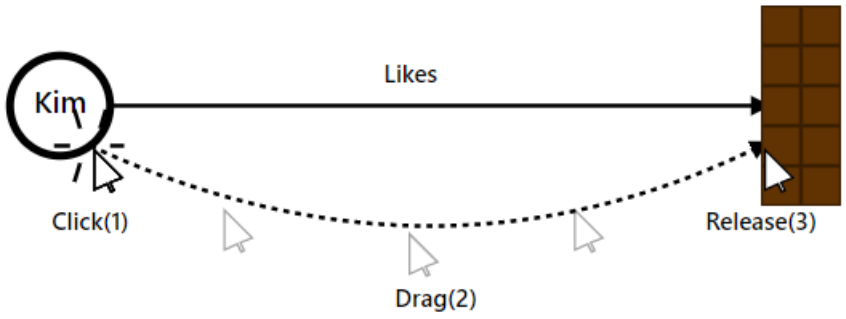


Figure 4.4. Example for the click-and-drag interaction.

Let us start by decomposing this interaction with a basic scenario. In our example, the end-user wants to create a new reference from a concept named *Kim* to a concept named *Chocolate*. The reference is represented by the *Likes* concept, as seen in Figure 4.4. First the end-user clicks on *Kim* (1) and holds the click. Then, she drags the mouse to put it

on *Chocolate* (2). When she reaches the projection, she releases the click, and the edge is created (3). The analysis will consider that these steps take place in a projectional editor.

Click (1). The initial click has multiple implications. Since the subsequent steps will be used to set the target attribute of *Likes*, the concept *Likes* has to already exist in the ASG. Its creation may be the result of a click, which means that this effect must be directly specified in the projection (**Req #4**). Another option is that there is a button somewhere in the editor to select the type of concept to create during the interaction (like in mbeddr).

Drag (2). When the user holds the click, there is no effect on the ASG. Indeed, the value of the target of the connector will only be set in (3). While dragging the mouse, the user might hover the cursor over other projections. Because of **Req #4**, this interaction should be defined solely for the case when a click is being held. Creating such constraints requires a well-structured architecture to define and manage interactions with the projections.

Release (3). Finally, the result of the release is a critical point. If the user misses the target or makes an invalid selection, the *Likes* concept must be removed. The connector cannot be rendered, so its target cannot be edited anymore: there is no projection. This syntactical error cannot be present in the ASG. Considering that the click-and-drag interaction can be used in other contexts (e.g., moving code parts in a textual editor) this result may not always be desired.

This analysis only takes into account this specific context, but the click-and-drag interaction may need to be considered differently in other editors. In addition to defining the projections, the language engineer needs a global understanding of their structure to create specific behaviors that result from the interactions. However, a requirement can emerge from this analysis. When considering Step (3), missing the target can be managed differently on the concept side, but the representation will always come to the same result: the connector is not rendered. This leads to an additional requirement that states that ***the entire context of a relation must be resolved before connecting projections*** **(Req #6)**. Respecting this rule ensures that no semantically incorrect edge can be visible in the projections.

4.2.2. Projection Shadows

For most projectional editors, the click-and-drag interaction still requires more specifications [**9**]. Currently, Gentleman does not support this interaction. To render edges, we implemented another solution that can be generalized to graphical and projectional

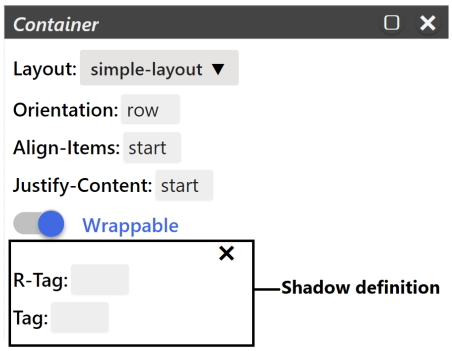


Figure 4.5. Shadow on a container-based projection in Gentleman.

editors. Each projection is related to a single concept, but multiple projections can be assigned to a concept. We use this principle to create *projection shadows*. Because connectors require information on their source and target concepts, setting these values can be realized in an alternative projection. This interaction-based projection can be textual, container-based, or even graphical. Once the values are known, the editor renders the edge. The interaction-based projection only needs a parameter to identify the connector.

Implementation. In Gentleman, the language engineer can assign a shadow to a projection. A shadow has a tag to identify the connector projection, and an *r-tag* to find the layout with its source and target, as seen in Figure 4.5. Once the connector is created, it waits for these values to be set to draw its path.

4.2.3. Drawing the connector

Once the connector can be rendered, the final step is to draw it. Specific editors can apply constraints on its path. For example, some language engineers may prefer straight lines whereas others focus on curved splines or bending points. Direction can also have an impact on the visualization, so she would use directed-edge. Managing these requirements can become costly considering a single edge projection. Implementing drawing instructions for each projection is not a time-effective solution. Hence, each drawing method should have its dedicated connector projection.

Implementation. For each graphical layout available in Gentleman, we created a dedicated connector projection. This helps managing edge-crossing and overlapping issues. A specific projection, the *Multi-Edge*, can be used when the path needs to reach more than two points. Each edge can be styled with CSS, display a direction, and declared with a decorator, a projection of another concept that is placed on the connector. These different projections

are visible in Table 4.1.

Projection	**Supported concepts**	**Usage**
Force-Edge	User-defined concept with attributes for the source and target.	The *Force-Edge* can be used in graphs. An attribute specifies if the path should be a straight line or a curved spline.
Anchor-Edge	*Reference*	The *Anchor-Edge* is registered in an *Anchor-Template* to find its source projection. It is rendered when the end-user sets the value of the *Reference*. By default, lines are straight, but the language engineer can set a specific path for self-references.
Multi-Edge	*Set*	The *Multi-Edge* covers the projections of the different items of a *Set*. The language engineer can modify the *meet* attribute to specify how the different points should be connected. The connector is rendered when the set has two or more elements.

Tableau 4.1. Connectors in Gentleman.

Chapter 5

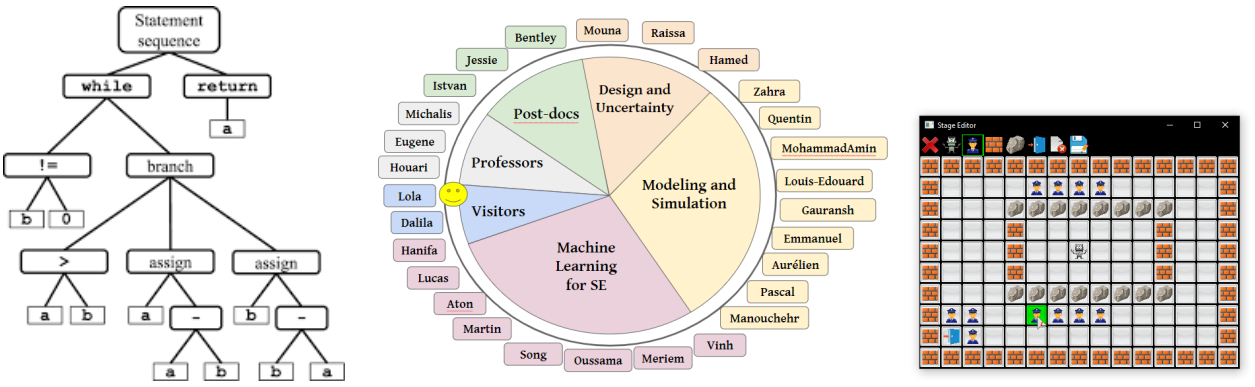
Layout Management

The coordinates and dimensions of a projection should be managed by their layout (**Req #2 and #3**). Graph drawing techniques are proven to be quite efficient to render models while considering visual aesthetics [**34**]. As the end-user interacts with the model, the ASG evolves and the projections have to adapt to it. ***Relying on simple layout algorithms that can be called periodically*** **(Req #7)** becomes a requirement for a projectional and graphical editor.

5.1. Creating layouts

The first step in order to define a layout is to think about the desired structure. Depending on the DSL, the computation disposition might be affected by different elements.

5.1.1. Analysis



(a) An abstract syntax tree (b) Participants in a meeting (c) A video-game stage editor

Figure 5.1. Different layouts associated with three models.

Figure 5.1 shows three different layouts applied on three models. Our extension of Gentleman only supports the creation of (a) and (c) since (b) requires the creation of mathematical

notations to generate the circular disposition, a feature not available on the editor. First, there is an abstract syntax tree. The disposition of the elements follows a vertical flow. An element is located below its parent and above its children. The second editor represents participants in a meeting. Each person is represented around a circle. In the circle, subdivisions display the area of research of the participants. The disposition of the participants follows a regular evolution around the circle. Finally, the third model represents a video game stage. The general disposition is represented by a grid-like structure. Each character can take a place in a dedicated cell. Coordinates are related to the cell, with very little interest in the presence of other model elements (aside from non-navigable areas or superposition). Each of these models uses a specific layout that focuses on different criteria. In (a) and (b), the coordinates of a node are impacted by the position of other elements. In (b) and (c), positions can be estimated easily because a vertex can only be positioned at some precise locations. Generalizing these various structures into a single layout with multiple parameters would be difficult. As the algorithm that manages the disposition can become very sophisticated, creating new implementations and importing them into the projection would become a time-consuming process. The solution we propose is to offer different families of layouts, in addition to the force layout, with very limited and specific parameters to meet the requirements of each projection model.

5.1.2. Predictable coordinates and size

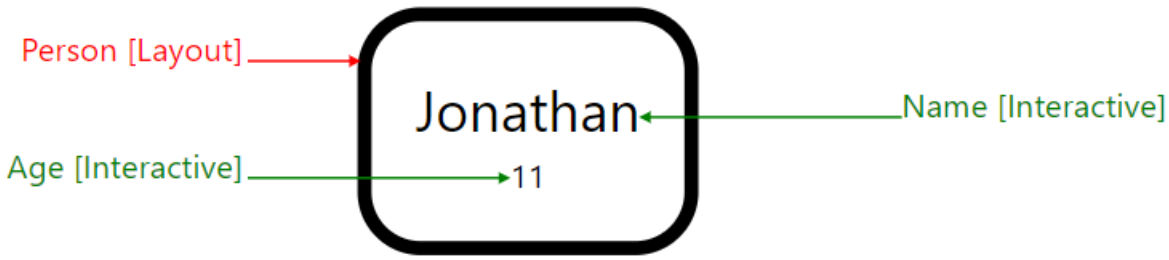


Figure 5.2. The projection for a person with a *Decoration-Layout*

The elements contained in the different layouts presented in Figure 5.1 are, to some extent, impacted by the presence of other concepts. This is not always the case. For example, let us consider a very simple projection for a concept named *Person*. A person has a name and an age. The projection for the concept is represented in Figure 5.2. First, we have a layout represented by a black roundtangle. This layout manages the disposition of two interactive projections: the name and the age. The coordinates of these two elements can be predicted as they are not affected by other concepts. If this projection was contained

in a more complex layout, say a graph to show connections between multiple persons, then the only coordinate modifications will be applied to *Person*. Attributes that are contained in the layout would not change. This structure is very similar to the ones found in textual projectional editors. In MPS, when creating the projection for a concept, the language engineer is presented with a tabular projection. She can then place attributes or static projections (e.g., text or spaces) in the different cells to organize the layout. For graphical projections, the same procedure can be adapted by considering a layout as a blank page. The language engineer can then attach elements with specific coordinates and sizes. We define this layout as a *Decoration-Layout*. It is only suited for concepts with atomic attributes, hence a concept with a relatively fixed amount of information.

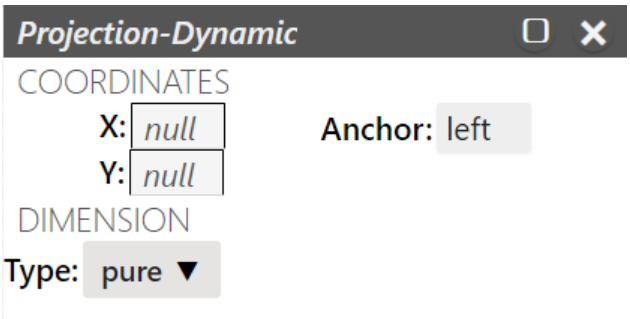


Figure 5.3. Coordinates and size definition in the *Decoration-Layout*.

Implementation. The default graphical projection in Gentleman is the *Decoration-Layout*. It has a size and a background. Each projection it contains is declared with specific coordinates and dimensions, as seen in Figure 5.3. When the layout is rendered, it starts by managing the size of the element. They can have three possible values: *pure*, *fixed* and *absolute*. An element with a dimension set on pure is directly added to the layout without modification. Fixed size uses an integer to set the width of the projection. When added to the layout, the editor computes the dimensions of the projection using a method returning the size of an element relative to the viewport and adapts the height using the ratio between the integer and the estimated width returned by the function. To prevent any problem, the editor uses the *viewBox* attribute of the SVG. It allows an SVG element to define its coordinate system. Hence, the *viewBox* uses the dimensions returned by the function while the width and height of the SVG are set according to the *fixed* integer. Finally, *absolute* dimensions dictate directly the width and height of the projection. After managing the size, the layout sets the coordinates of the projection. Because it has information on its width and height, the layout can center the element if needed. The different parameters of the *Decoration-Layout* are visible in Table 5.1.

Attribute	Type	Usage
Dimensions (*optional*)	Absolute dimensions	Height and width of the layout. If dimensions are not defined, the layout wraps the elements it contains.
Coordinates (*optional*)	Coordinates	Position can be directly defined in the layout if it is contained in another graphical projection.
Background (*optional*)	String	Imported SVG file used as a background for the layout.
Shape (*optional*)	Shape	SVG element created in Gentleman using the *Shape-Editor*. If the shape is defined, the background attribute of the layout is ignored.
Content	Set of graphical components	Collection of elements contained in the layout.

Tableau 5.1. Parameters of the *Decoration-Layout*.

5.2. Graph-based Layout

Multiple graphical representations use graph structures to display the elements they contain (e.g., class diagrams, statecharts, mind maps). As projectional editing requires automatic positioning, finding the optimal coordinates for a projection can become a challenge. Vertices are impacted by the presence of other elements to avoid superposition and edge-crossing. In this section, we briefly present the principles we adopted in Gentleman by creating a *Force-Layout*.

5.2.1. Force-Layout

To create a graph-based layout in Gentleman, we considered multiple approaches. We started by exploring different graph-drawing techniques to find a simple layout management algorithm that would be adapted to an editing environment where elements can be added and removed. The first idea that emerged was to create a *Grid-Layout* that takes inspiration from the orthogonal drawing techniques. It consists of a grid with an initial number of squares that could evolve after the addition of a new vertex in the graph. Edge-management uses a shortest path algorithm that prevents any edge-crossing and adapts the grid if needed. However, we realized that our implementation could be improved but it implied taking more time to focus solely on graph-based drawing. To make sure we had the time to propose implementations of all our layouts, we decided to use a library to manage the computation of the coordinates of the vertices and extend the algorithm to add new features. That is why we decided to use the D3.js[1] force simulation.

[1]https://d3js.org/

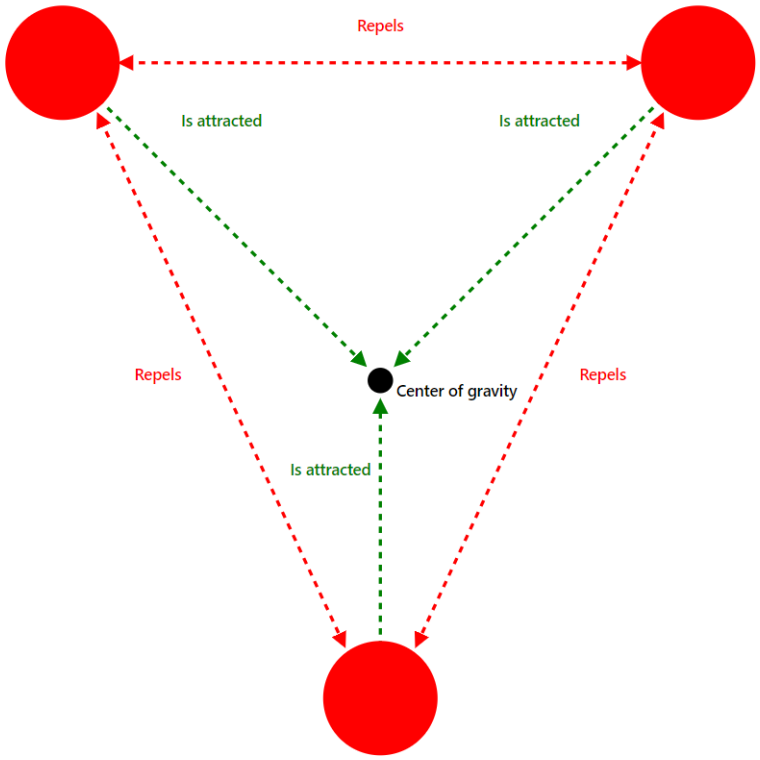


Figure 5.4. Description of a force-directed layout.

5.2.1.1. **Force-directed layout**. Drawing techniques often take inspiration from physical models [**33**]. In a force-directed layout, nodes are attracted to a center of gravity, as seen in Figure 5.4. In addition, actions of repulsion/attraction are applied between the nodes. The goal is then to minimize the general energy of the system to generate optimal coordinates. Using a force layout makes a lot of sense for the creation of a graph. The center of gravity allows keeping the model centered while the repulsion prevents any superposition. Attraction can be used to group strongly connected vertices in a specific area of the graph. Because of the requirement of visual aesthetics (**Req #5**), keeping the general organization of the nodes implies that the graph should not be redrawn after each addition/suppression of a concept. A solution is to consider previous coordinates every time the algorithm is re-run.

5.2.1.2. **Approaches**. The computation of a force-directed layout can be realized with different approaches depending on the representation of the energy in the system [**63**]. Popular models either focus on a fixed distance between every pair of nodes or modularity by creating different clusters in the layout. The first one only uses actions of repulsion whereas the generation of modules uses attraction to encourage the concentration of the connected nodes in a restricted area. However, most approaches usually consider nodes as a single point

in the graph, with little regard to properties like the shape or the size. Since this process might encourage overlapping issues, post-processing techniques can be applied to clean the layout once the graph is rendered [**33**].

5.2.1.3. **Implementations and parameters**. Force-directed layouts are very popular in network visualization. Tools like Graphitti [**64**] or CellNetViz [**65**] use D3.js to compute the initial coordinates and apply their own post-processing methods to adapt the vizualisation to their domain. Since the layout is tailored to their respective purpose, they offer specific parameters to adjust the representation. When considering the general approach of force-directed methods, three important parameters are required. First, the position of the center of gravity. Usually, it is placed in the middle of a graph with predefined dimensions. The second parameter is the value of the action of repulsion. Ensuring that nodes repels each other minimizes overlapping issues. Finally, deciding on length of links guarantees an even distribution of the graph. In some approaches, this parameter can be replaced by actions of attraction between two adjacent nodes to create a cluster [**63**]. However, the idea remains the same: connected elements should be treated differently. When the graph is considered as dynamic (e.g., new nodes or link can be added), additional settings custom the speed of the placement of the nodes or limit the duration of the algorithm by stopping it after a certain amount of time.

5.2.2. Adaptation on the projection

As explained before, force-directed methods usually consider nodes as points in the graph. Projections may have various dimensions which may lead to overlapping issues- [**33**]. Dealing with a cluttered graph often requires layout adjustment methods that can impact the end-user mental map of the model [**66**]. Graphical projections may differ from one editor to the other. Having the editor adapt itself to the graphics to apply specific adjustments may become too difficult. The solution we propose is to describe all possible projections with basic easy-to-manage shapes.

Describing the projections. The layout sets the size of a projection. Imported graphics may have predefined dimensions, thus the editor should be able to perform basic operations to adjust the width and height until the desired criteria are met. In classical drawing editors like Paint [**67**], images are contained in a bounding rectangle. Modifications of the dimensions are performed by increasing or decreasing the width and height of this rectangle. A similar approach can be applied to describe projections in the layout. If the algorithm only considers node points, post-processing methods could consider them as rectangles. Since the layouts manage the dimension of a projection, placing an element then only requires setting the coordinates of the rectangle, with potential additional computation

(e.g., centering a projection requires subtracting half of its width to x and half of its height to y). Cases of superposition are then only considered between different rectangles for all editors. The limit of this approach is that it might become too general for some DSLs with complex graphics, as it ignores the visual borders of a projection and the rectangle might appear as too large. However, it guarantees a general treatment that can be used in different contexts.

Overlapping management. Managing overlapping issues requires a re-organization of the nodes coordinates. Each projection in the layout is centered so that the repulsion is evenly distributed around it. Two rectangles that occupy a shared space are then considered to overlap. To prevent such cases, coordinates have to be modified. Because both elements are attracted to the center of gravity, moving only one projection would not make much sense as it would decentralize the layout. To ensure an even distribution of the projections, the offset necessary can be found using the minimum between the width and the height of the overlapping area. As only the two projections are going to move, some new cases of overlapping with other elements can occur in the layout. To prevent having to resolve too many issues, modifications of coordinates should always result in a greater distance between the projection and the center of gravity. The overlapping resolution will then only result in a spread of the layout.

5.2.3. Edge-management

Limiting edge-crossing is a very important criterion for many graph drawing techniques-[**68**]. As the graph created by the end-user might be non-planar, avoiding this problem becomes impossible [**69**]. In a force-directed layout, edge-crossing can be discouraged by using dummy nodes placed on the edges to create additional repulsion. The resulting layout then returns the position of the edge accordingly to the coordinates of the nodes. As projections are considered as centered rectangles by the layout, connecting two points leads to new concerns. Overlapping can be managed by readjusting coordinates, but edges also need treatment to avoid crossing a projection. A simple solution could be to consider layers, with edges being rendered under the projections. This causes a problem when the edge drawn needs to indicate a direction. Moreover, only connecting the rectangle might lead to cases where the edge is seen as not reaching the actual target shape. In addition to overlapping resolution, a second adjustment needs to be applied to detect the borders of the projection. As the language engineer creates the projection, she can directly specify the shape that will be considered as the border. Because projections are centered, estimating the border then only consists of finding the intersection point between the direct line joining the source and target centers and the designated shape. This computation can be easy to do for ellipses and

rectangles. For more complicated structures, an alternative solution is to specify directly anchor points where the edge can connect.

5.2.4. Implementation

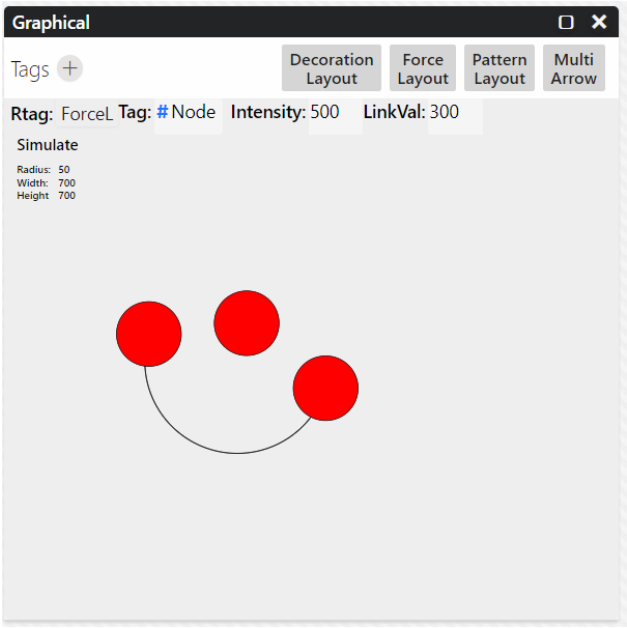


Figure 5.5. Definition of a *Force-Layout* in Gentleman.

The implementation of the Force-Layout in Gentleman uses a force simulation created in D3.js, a library to manipulate documents based on data [**70**]. The definition of the layout in Gentleman and the corresponding simulation are shown in Figure 5.5. Each projection in the layout is registered in the data as nodes of the simulation with information on their shape and dimensions. A tick function is periodically called to update the coordinates, using the *intensity* parameter as the repulsion, and adapts their value to ensure that the projection is visible in the layout. As the simulation only focuses on computing positions, we added post-processing functions for collision and border detection, using the methods described previously. Edges are considered links in the simulation and are modified every time the coordinates are updated. The *linkVal* attribute sets their length. To allow the end-user to create multiple edges with the same source and target, the connectors are represented with arcs. This ensures that every SVG path is not superposed. A summary of the layout parameters and their purpose is shown in Table 5.2.

5.3. Pattern-Layout

The computation of coordinates can sometimes follow a sequence. In the *Pattern-Layout*, the language engineer defines a pattern (a projection) that will be repeated in the layout.

Attribute	Type	Usage
Intensity	Number	Intensity of the actions of repulsion. As D3.js interprets an intensity greater than 0 as attraction, the layout only considers negative values.
LinkVal	Number	Fixed length of a link in the simulation.
Dimensions	Absolute dimensions	Size of the layout. The center of gravity is always placed in the middle ($x = width/2$, $y = height/2$).
Item	Tag	Identifies the projection of a node.

Tableau 5.2. Parameters of the *Force-Layout*.

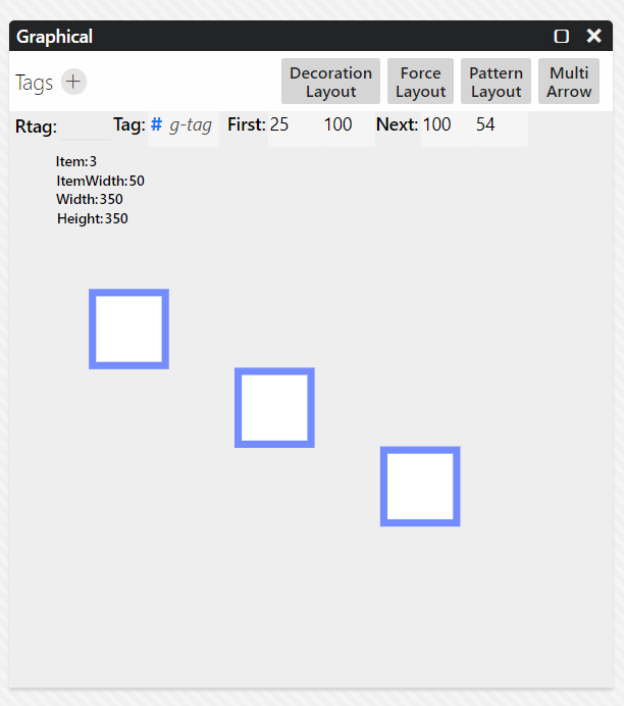


Figure 5.6. Definition of a *Pattern-Layout* in Gentleman.

Coordinates are established using an anchor. An anchor is composed of two parameters: the initial position and the operations to apply to get to the next position. The initial positions are defined relatively to the layout, so the language engineer can visualize the beginning of the sequence. The second parameter is defined as the evolution of these coordinates. The x and y values given by the language engineer are directly added to the previous coordinates when the layout computes the next position. Unlike MPS [**11**], Gentleman does not support mathematical notations yet, so anchors can only describe arithmetic sequences in the form of $x_{new} = x_{\text{init}} + i \times x$ and $y_{\text{new}} = y_{\text{init}} + i \times y$ for the i^{th} projection in the sequence. As the sequence can be hard to visualize with textual notations, the editor offers a simulation of the estimated rendering of the layout as seen in Figure 5.6. When created, each projection

is registered in the layout with its corresponding coordinates. After removing an element, the layout moves all the following projections to the previous coordinates to keep the model compact. Attributes of the layout are shown in Table 5.3.

Attribute	**Type**	**Usage**
Saturation (*optional*)	Number	Maximum number of patterns that can be displayed in the layout.
Template (*optional*)	String	Allows the layout to register in an *Anchor Template*.
Pattern	Anchors	Arithmetic sequence that dictates the evolution of the coordinates.
Item	Tag	Identifies the projection of the pattern.

Tableau 5.3. Parameters of the *Pattern-Layout*.

5.3.1. Saturation

The *Pattern-Layout* only represents a collection of elements. This set of projections can be contained in another layout that adds background or manages other types of elements. Adding new projections can lead to a situation where this containing layout becomes too small. If the concept has a constraint on the number of items it can contain, then no adaptation is required, because of the constraints on the ASG. For unbounded sets, two options are offered in Gentleman. First, the language engineer can decide that the pattern-layout must saturate. Saturation is defined by an integer representing the maximum number of projections that can be displayed in the layout. When the maximum capacity is reached, the layout asks the editor to create another layout to manage the rest of the elements. This creation may require an additional layout for containment that can be identified using an *r-tag*. The second option is to define a transformation in the containing layout. This transformation uses a marker *data-augment* that contains information on the operation that will be performed on a specific property. Every time the pattern-layout creates a new projection, it checks if augmentation is required. The containing layout then changes if needed. Currently, Gentleman only supports modification of the width and height of an SVG element.

5.3.2. Anchor Template

Anchor Template is a string attribute that can be added to a pattern-layout. As the layout can be created multiple times in the editor, because there are multiple instances of its corresponding concept, the language engineer might decide that all patterns follow a shared sequence. When created, each pattern layout registers itself in the template. Each operation

is then transferred to the other projections. Elements in the template are not related to the same instance of a concept, but they share the sequence. Take the example of three pattern-layouts A, B, and C that share a common anchor template. If a new pattern is created in A, then B and C will compute their next coordinates to be at the same level. After removing an element, all of the positions in the different layout decrease. Using an anchor template can become very useful when the end-user wants to create references between concepts of different sets. The *Anchor-Edge* can be placed on a pattern and directly find its target coordinates using the template.

5.4. Tree-Layout

Hierarchical representations can be very important to reflect the underlying architecture of a model. In the graph layout discussed in Section 5.2, the estimation of projection coordinates only focused on overlapping issues and an even distribution of the elements in the layout. In a tree structure, the disposition can carry additional semantical information on the model. Multiple algorithms exist to generate different types of tree layouts such as the radial view [**71**] or the balloon drawing [**72**]. For our implementation we focused on the hierarchical view [**73**] using a level-based approach [**74**].

5.4.1. Conceptual requirement

Creating a general tree layout requires constraints on the hierarchical structure of the metamodel. As the organization of the elements is level-based, we need a root concept that the end-user can use to make the tree evolve. A good example is the mindmap used in Section 2.4.2. The model starts with a *MindMap* containing *CentralTopic*. The central topic can then be composed of *MainTopics* with *SubTopics* that can also contain other subtopics recursively. The ASG of this model is actually a tree. To create such a structure, the important concern is to define the concept that will be represented by the layout. Indeed, it will manage the disposition of projections that are related to concepts located in various parts of the tree, with different depths. A possible approach is to create subtrees that only consider a root and the direct children of the concept. However, maintaining a compact structure and avoiding overlapping issues requires that a subtree knows the position of every node in the layout before computing the coordinates of its child elements. The adopted solution is to directly define the tree layout on the root concept, and then represent every element that needs to be a node as a projection directly located in the layout.

The main challenge with this approach is to define the position of an element in the structure as the concepts may be located in different areas of the ASG. Because of the requirement for automatic positioning (**Req #1**), the editor has to identify the

different nodes of the tree. As the evolution of the layout is based on the addition of new concepts, we created a new trigger for our SVG-Buttons named "CREATE-NODE". This button has information about the tree, using an *r-tag*. Using the hierarchical structure of the metamodel, this static projection can be added to each node that can have children in the tree. The projection is directly representing an attribute of the concept. Whenever the end-user creates a new element, the button finds the closest parent concept present in the tree and sends the new node to the layout using its *r-tag*. After receiving the new concept, the layout analyzes the location of its parent and computes its position. Deleting a node implies its attributes are also removed from the ASG, so the subtree is entirely removed from the layout. Parameters of the layout are shown in Table 5.4.

Attribute	**Type**	**Usage**
TreeId	String	Identifier of the tree. Ensures that new nodes are added to the correct layout.
Orientation	String	Direction of the tree.
Depth	Number	Space between two levels of the tree.
Item	Tag	Identifies the projection of the nodes.

Tableau 5.4. Parameters of the *Tree-Layout*.

5.4.2. Implementation

In Gentleman, the *Tree Layout* uses D3.js to compute the coordinates of the nodes. The data of a node contains information about its concept so that the layout can directly identify where to place its children. When defining the projection, the language engineer can specify multiple attributes she can visualize with a dedicated simulation. The *orientation* is a string that dictates the flow of the tree. The computation adapts the coordinates given by D3.js to organize the tree in the corresponding direction. The *depth* specifies the distance between a child and its parent. Coordinates are adapted to become the factor of the depth of a node and this value. As elements are directly connected when rendered, there is no edge-dedicated projection for the *Tree Layout* layout in Gentleman. The language engineer can directly apply style on the path that will be drawn each time the end-user adds a new child element.

To illustrate the implementation, we recreated the mindmap editor with graphical projections in Figure 5.7. The *MainTopic* of the mindmap is used as a root for the tree layout. As concepts in the tree can be composed of numerous *CentralTopics* or *SubTopics*, each node projection contains an SVG-Button (rendered as as circled cross in Figure 5.7) with the "CREATE-NODE" trigger declared with the *r-tag* of the layout. Each topic has a name that can be directly edited in the node.

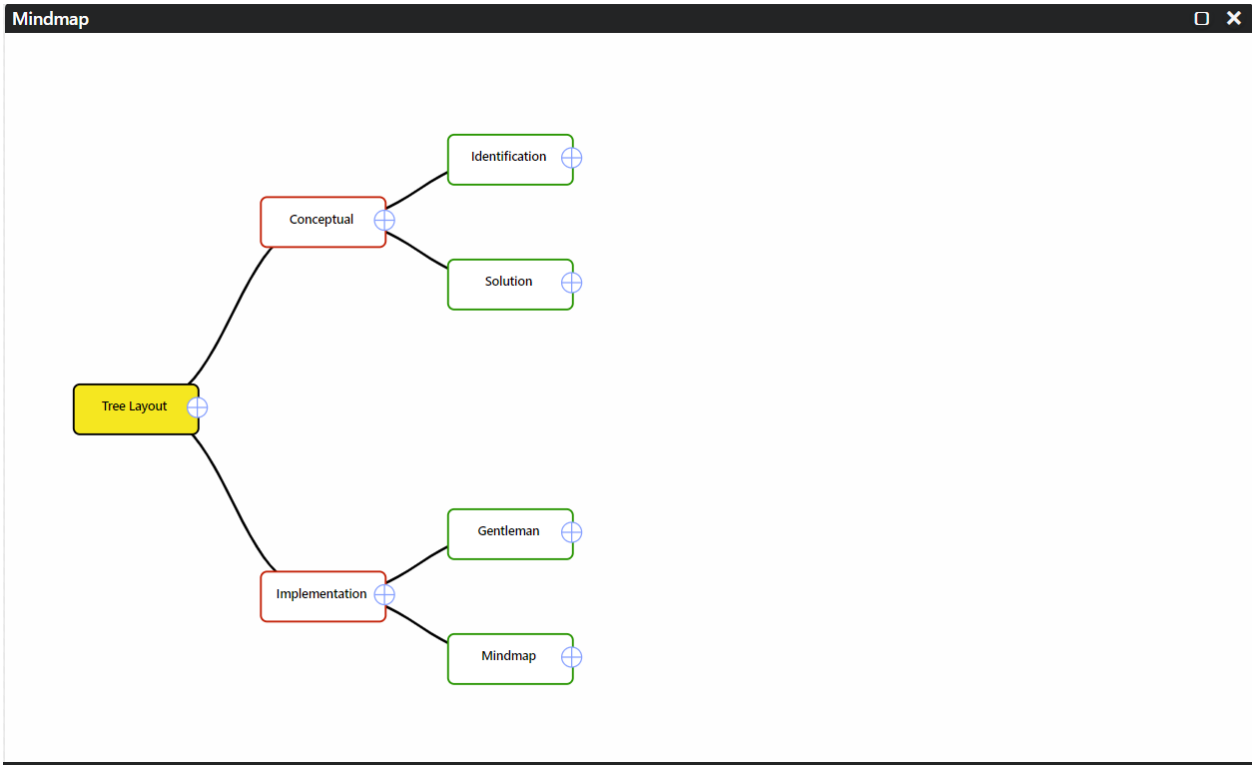


Figure 5.7. The graphical mindmap editor.

5.5. Summary

The differents layout we created are compiled in Table 5.5.

Layout	Purpose
Decoration-Layout	Default layout. The elements it contains have predictable size and position.
Force-Layout	Graph-based layout. Uses a center of gravity, actions of repulsion, and post-processing methods to place the nodes and links.
Pattern-Layout	Predictable evolution in the coordinates of the projections contained in the layout. Can be registered in an template to share the sequence with other layouts.
Tree-Layout	Hierarchical tree. Has to be defined on the root element of the tree.

Tableau 5.5. Available layouts in Gentleman.

Before discussing our application examples, we also compile the requirement established in Chapter 3 and Chapter 4:

- **Req #1**: the focus should be to keep the interaction in the projections, hence, in the graphics.
- **Req #2**: the positioning of the object must be directly managed by the editor.

- **Req #3**: the editor must be able to automatically compute dimension modifications.
- **Req #4**: each interaction point must be designed with its own projection.
- **Req #5**: visual aesthetics must be considered when designing a layout.
- **Req #6**: the entire context of a relation must be resolved before connecting projections.

Chapter 6

Application Examples

To validate the applicability of this new graphical projectional editors, we showcase how to use it for three distinct graphical DSLs: statecharts focusing on a hierarchical graph structure, UML sequence diagrams repeat patterns and complex connection, and music sheets using a specific notation with precise positioning. Since our extension is in its early stages, some improvement may still be done for each examples.

6.1. Statechart Editor

Statecharts are a visual formalism created in order to represent reactive systems [**2**]. It can be compared to a finite automaton, with different states connected with transitions [**75**]. Statecharts allow representing the effects of sequences of events on complex systems. They can be used for many purposes such as visualization, execution, or debugging. Multiple tools can be used to create and interact with statecharts such as Yakindu [**76**] or SCXML [**77**].

Statecharts offer a great challenge when considering a projectional editor. The graph-based disposition of the different states and transitions requires specific attention to visual aesthetics to ensure that each concept is easy to visualize. Moreover, the presence of different regions in the model implies that multiple graphs can be displayed simultaneously in the editor. To generate an editor with Gentleman, we created an adapted metamodel to ensure that each concept could be represented with a projection. The projection model is available on github.[1]

6.1.1. Traditional Editors

Editors like Yakindu[2] or the Qt[3] statechart editor often offer two different views of the model: a canvas for visualisation and a textual area for advanced editing. To create new

[1] https://github.com/geodes-sms/gentleman/blob/master/models/statechart-model/projection.json
[2] https://www.itemis.com/en/yakindu/state-machine/
[3] https://doc.qt.io/qtcreator/creator-scxml.html

states and transitions, the end-user selects a concept from a panel and places it directly on the canvas. She can directly adjust the size and position of the elements or modify the path of a transition. In addition, the dimensions of a state automatically adapt to cover the different actions and regions contained in the representation. Since these editors also offer support for executing and debugging statecharts, the textual area contains information on the different variables and an overview of the structure of the statechart. Errors of conception are signaled to the end-user so that she can fix them manually. These editors offer very little support for automatic layout. In the QT editor, unnecessary edge points are removed when a new transition is rendered, but the user can still modify the path. For YAKINDU, the orientation of the subregions in an orthogonal state can be automatically adapted depending on their content, but the disposition of the states is not affected.

6.1.2. Metamodel

```
1 concept root Statechart : {
2     attributes:[
3         { name: "name", type: string}
4         { name: "entry", type: DefaultState},
5         { name: "states", type: set, accept: State},
6         { name: "transitions", type: Set, accept: Transition}
7     ]
8 }
9
10 prototype State: {}
11
12 prototype PseudoState: State {}
13
14 concept DefaultState: PseudoState {}
15
16 concept ChoiceState: PseudoState {}
17
18 concept HistoryState: PseudoState{
19     attributes: [
20         { name: "deep", type: boolean}
21     ]
22 }
23
24 concept BasicState : State {
25     attributes:[
26         { name: "name", type: string},
```

```
27          { name: "entry", type: set, accept: string},
28          { name: "exit", type: set, accept: string}
29      ]
30  }
31
32  concept CompositeState : BasicState {
33      attributes: [
34          { name: "states", type: set, accept: State},
35          { name: "default", type: DefaultState}
36      ]
37  }
38
39  concept OrthogonalState: BasicState {
40      attributes: [
41          { name: "regionA", type: CompositeState},
42          { name: "regionB", type: CompositeState}
43      ]
44  }
45
46  concept Transition {
47      attributes: [
48          { name: "from", type: reference, target: State},
49          { name: "to", type: reference, target: State},
50          { name: "description", type: string}
51      ]
52  }
```

Listing 6.1. Statecharts metamodel.

Listing 6.1 represents the metamodel of the DSL. The root concept is a *Statechart*, composed of a set of *States* and a set of *Transitions*. The *State* concept is defined as a prototype that is extended by *PseudoState* and *BasicState*. There are three pseudo-states in the metamodel. *DefaultState* represents the entry point of each region of the statechart. *ChoiceState* can be used to create decision branches. Finally, *HistoryState* allows remembering the last active state in a region. The *bool* attribute can be set to *true* to keep track of the status of the nested states. The second concept extending the *State* prototype is *BasicState*. It has a name and entry/exit actions represented as strings. This concrete concept is also extended by *CompositeState*, a state that encapsulates a sub-statechart to enable hierarchical nesting, and *OrthogonalState*, representing parallelism. In our metamodel, we only represented the latter with two regions to keep a small representation in the projections, but more could be added using for example the *Pattern-Layout*.

Transitions are composed of three attributes. *from* represents their source and *to* their target. In addition, a transition has a *trigger*, a *guard*, and an *effect* all captured in a single string. They allow the specification of some well-formedness constraints. For example, a microwave starts heating food when the "Start" button is pressed (*trigger*) and the door is closed (*guard*). As a result, the temperature of the food increases (*effect*). Since these attributes do not have an impact on the graphical editor, we omit them and only consider a simple string called *description* as an attribute of a *Transition*.

6.1.3. Projections

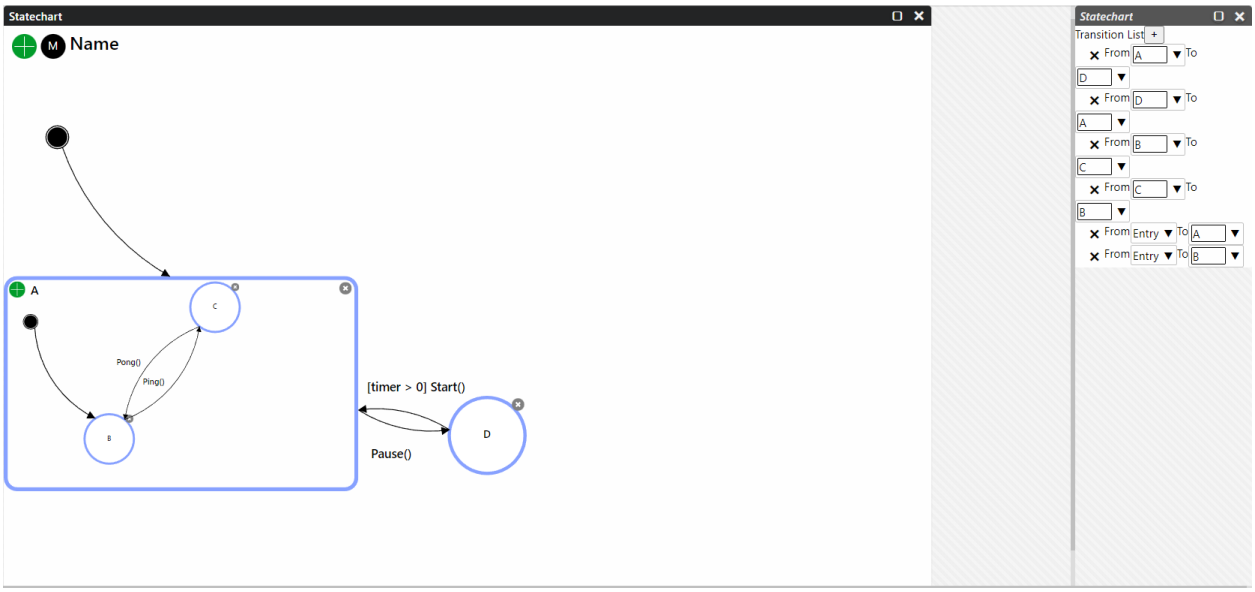


Figure 6.1. The statechart editor.

6.1.3.1. **Statechart**. The statechart is represented by a decoration layout visible in Figure 6.1. Indeed, each attribute has a predefined position and size in the projection. First, we have the name of the statechart represented in the top left corner as a text field. Two buttons are available next to this projection. The green one can be clicked to create a new state. This projection is directly related to the states attribute. The second button opens an alternative view of the statechart shown in the right part of Figure 6.1. This container-based projection can be used to create an manage the different transitions in the model. The two buttons and the name are located at the top part of the layout to create the impression of a panel. Indeed, the rest of the layout is dedicated to the representation of the states. Since this attribute is a set, it is defined with a dedicated projection to manage the disposition of its items (**Req #2**).

6.1.3.2. **Set of states**. Each set of states has two projections: a button similar to the one described in the previous paragraph and a force layout that displays the elements.

```
function computeCircle(container){

    /* Finding the SVG element in the projection */
    const target = container.querySelector("[data-shape]");

    /* If not found, return nothing. The calling function will raise an exception */
    if(isNullOrUndefined(target)){
            return;
    }

    /* Estimation of the size of the projection in the layout */
    const rectItem = target.getBoundingClientRect();

    /* Estimation of the size of the shape in the layout */
    const rect = container.getBoundingClientRect();


    /* Computation of the scale */
    ratio = Number(target.getAttribute("r")) / (rectItem.width / 2);

    /* Returning the radius */
    return rect.width * ratio / 2;
}
```

Listing 3. Estimation of the dimension of a circle.

Indeed, the position of each state will be impacted by the presence of other elements, which motivated the use of a graph-based layout. The different force layouts apply the repulsion between the states and their attraction to the center of gravity. Each element is represented in the data used by the D3.js force simulation with its respective height and width. If a projection changes, these values are updated. Because it has information on the different state coordinates, the layout also manages the rendering of the different transitions. To ensure that transitions connect to the border of a state, the data also registers the shape of the states by looking for a vector with a *data-shape* marker. Computing the shape uses the estimated dimensions of the vector in the *viewPort* and translates them into the layout coordinate system. This process is shown in Listing 3 with the example of a circle.

6.1.3.3. **State**. A state is defined as a prototype concept in the metamodel. Rather than creating a dedicated button for each concrete value, we represent the selection as a choice-field (**Req #1**), as seen in Figure 6.2. This projection displays the possible values for a concept in a tabular form. Each projection shown in the field is identified with the "choice" *tag*. Clicking on one of them sets the value of the concept. Since a state can be removed from the set, a button is attached to the projection. Depending on the value of

the concept, a mapping describes its position to ensure that it is always optimized for the projection (**Req #5**). Once the selection has been made, it cannot be changed to limit the presence of interaction-based buttons in the projection. If the end-user creates the wrong type of state, she can remove it and create a new one.

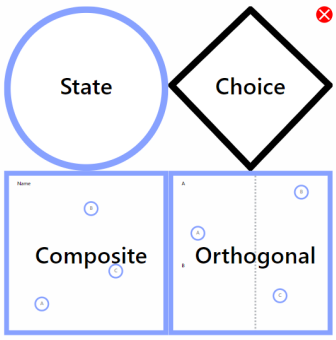


Figure 6.2. When instantiating an abstract prototype state, the end-user has to choose the concrete concept that extends this prototype.

6.1.3.4. **Different type of states**. The concrete and choice states are the two concepts that do not contain regions. To offer diversity, and test the estimation border function of our force-layout, they are represented by two different projections. The concrete state is a decoration-layout with a circle as a background. The name is directly represented in the middle with an additional button to open a container-based projection containing its different actions. The choice state background is composed of a polygon. As the estimation of the borders may be harder for this type of SVG vector, the layout also has predefined anchor points. These anchor points are described directly in the polygon vector with a specific marker. Composite and orthogonal states contain subregions. To offer better visualization of the items of the set, they are created with a rectangle as a background. As they may contain elements that also have *subRegions*, a *viewBox* is applied to the concept to scale their different projections.

6.1.3.5. **DefaultState**. The entry state is represented with a basic projection inspired by Yakindu [**76**]. *DefaultStates* are not contained in the different sets of states present in the metamodel. Indeed, Gentleman does not offer support for such complex constraints on the set concept. Hence, they cannot be directly managed by the force layout used to represent these concepts. Each concept containing a region is represented by a decoration layout. These projections directly organize the entry point to place it in the top-left corner

of the layout.

(a) Container-based projection of *Transition.*

(b) Graphical projection of *Transition.*

Figure 6.3. Transition management in the editor.

6.1.3.6. **Transitions**. To represent *transitions*, two projections are defined in the model. The first one, shown in Figure 7.8(a) uses a container-based representation to allow modifications of the source and target attributes. The projection is a shadow (cf. Section 4.2.2) pointing to the common *r-tag* shared between the different force layouts in the model and uses its tag to create a *Force-Edge*. The force edge, shown in 7.12(b), can be drawn using the source and target attributes of a transition. Once the values have been set, the connector signals that it can be drawn to the active layout that has the specific *r-tag* (**Req #6**). The active layout is the last graphical projection the end-user has interacted with for a family of layouts. If the layout contains the projections of the nodes, the editor renders the connector. If not, it queries other layouts in the family to find the suited representation. After the layout computed the path of the edge, the editor displays the projection. Because transitions have a string attribute, a projection is added to the edge as a decorator and placed on its dummy node. In some specific cases, the source and target nodes can be located in different layouts. Since D3.js can only manage nodes and links in a specific SVG element, these projections are considered as *translinks*. *Translinks* apply no repulsion as they are not in the simulation. Every time the tick function is called, it analyzes the position of the source and target in the editor and translates coordinates in the layout *viewBox* system to create the path. *Translinks* allow for example connecting the different states to the default states.

6.1.4. Discussion

The strong asset of the Statechart example is that it can contain multiple graph-based layouts. These layouts are all managed with their own force simulation and can coexist in the editor. The algorithm that uses D3.js easily deals with any case of overlapping and the border detection is well adapted to any kind of shape. Constraints of automatic positioning and automatic sizing are respected, and the fixed length of the transitions allows the creation of

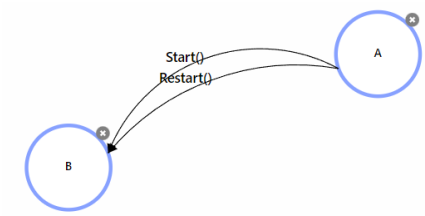


Figure 6.4. The issue of the dummy nodes.

clusters for strongly connected parts of the model. Since D3.js randomly assigns coordinates to new states, the disposition of the state is different for all models.

However, some improvements can still be achieved. First, the editor focuses only on the creation and visualization of the graphical structure of the statechart. Some concepts like events or variables are not represented in the metamodel. A second concern is that the management of transitions that have the same source and target is still not mature enough. Figure 6.4 demonstrates this statement with a very simple example. As a transition description is directly placed on a dummy node, the position of the text is considered fixed. Managing collision then becomes a more difficult task since changing these coordinates may result in a text being too far from its corresponding edge, making it harder for the end-user to understand the underlying structure. Finally, the composition of regions is possible in the editor by making the contained projections look smaller with a *viewBox*. This structure is required to ensure for example that a composite can contain another a composite. However, Gentleman does not offer yet a zoom-in interaction so projections may become hard to see. A solution that is currently being explored is to adapt the size of the layout to the elements it contains, which implies dynamically adapting the parameters of the simulation. The management of the different requirements in this example is described in 6.3.

Requirement	**Management**
#1	Basic editing activities such as the creation and suppression of the states are realized in the graphics. For the generation
#2	The force layout manages the disposition of states and transition.
#3	*CompositeStates* and *OrthogonalStates* currently have fixed dimension. An improvement would be to make the size of their layout adapt to the number of element they contain.
#4	This requirement is respected.
#5	The presence of dummy nodes on transitions discourages the possibility of edge-crossing. The computation of coordinates ensures that the model is compact without any case of overlapping elements.
#6	This requirement is respected.

Tableau 6.1. Requirements in the statechart editor.

6.2. Sequence diagram

UML sequence Diagrams focus on expressing the dynamic behavior of a system [**78**]. They represent specific scenarios by displaying the different actors and how they interact. They communicate with *messages* to invoke methods that send *responses* with a potential returning value. Alternative scenarios, options, and loops can be added with *fragments*. In [**79**], sequence diagrams are defined as complex structures that require specific layout management to emphasize their expressiveness. The general flow of the representation is mostly dictated by the ordering of different messages and blocs in the model. The specificity of this layout management makes the sequence diagram an interesting example for a projectional editor. The projection model is available on github.[4]

6.2.1. Traditional Editors

Most sequence diagram editors like the one found in Visual Paradigm[5] use editing activities similar to the one described in Section 6.1.1. However, some online tools offer support for automatic layout[6]. Once again, they use a textual and a graphical view to interact with the model. The end-user can move elements in the graphics to arrange the order of the messages and responses, but the visualization is automatically adapted to find the optimal position. Graphical representations are only rendered after verification of syntax correctness. However, the creation of new elements uses textual notation. This part of the editor uses a parser-based approach to signal the errors in the model, but the end-user has to fix them to render the graphical representation.

6.2.2. Metamodel

```
1 concept root Diagram: {
2     attributes:[
3         { name: "name", type: string },
4         { name: "lifelines", type: set, accept: Lifeline}
5     ]
6 }
7
8 concept Lifeline: {
9     attributes: [
10         { name: "objectName", type: string },
11         { name: "elements", type: set, accept: Element}
12     ]
```

[4] https://github.com/geodes-sms/gentleman/blob/master/models/sequence-model/projection.json
[5] https://www.visual-paradigm.com/
[6] https://sequencediagram.org/

```
13 }
14
15 prototype Element: {
16     attributes:[
17         { name: "order", type: set, accept: number },
18         { name: "fragment", type: reference, accept: Fragment}
19     ]
20 }
21
22 concept Message: {
23     prototype: Element
24     attributes:[
25         { name: "target", type: reference, accept: Lifeline},
26         { name: "content", type: string }
27     ]
28 }
29
30 concept Response: Message{}
31
32 prototype Fragment: Element {}
33
34 concept Loop: Fragment {
35     attributes:[
36         { name: "condition", type: string }
37     ]
38 }
```

Listing 6.2. Sequence diagram metamodel in Gentleman.

The main components of a sequence diagram are the lifelines. In our metamodel described in Listing 6.2, they are directly represented as an attribute of the root concept, the *Diagram*. Each lifeline is designated by a name and a set of *Elements*. *Element* is a prototype that is extended by the different concepts describing the behavior and communication of the system. *Messages* and *Responses* call functions located in a class represented by a lifeline. The reference is contained in their target attribute. *Fragments* represent conditional structures that affect the flow of messages. In our metamodel, we only represented *Loops*. Indeed, *fragments* have a very specific structure that requires a dedicated layout. We present in the following section a possible implementation that we still need to work on before being able to export this representation and generate for example options or alternatives.

6.2.3. Projections

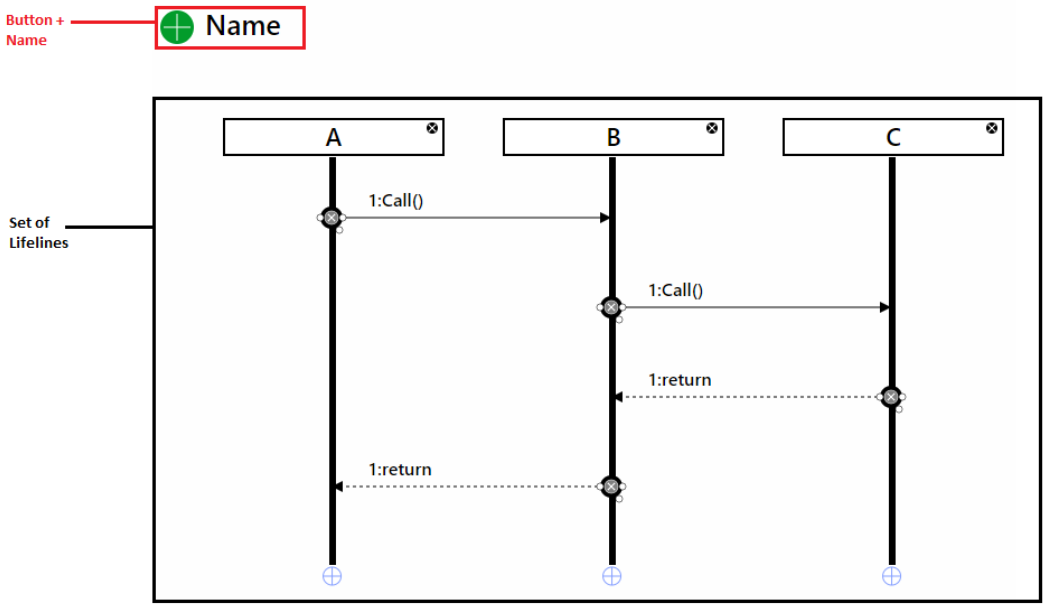


Figure 6.5. The decoration layout of the diagram.

6.2.3.1. **Diagram and set of lifelines**. The diagram is represented in the editor by a decoration layout shown in Figure 6.5. It contains a button to add a new lifeline and a text to set its name (**Req #4**). The position of the different projections in the editor is easier to predict than in a graph-based representation so no container-based projection is required for edge management. The rest of the layout is dedicated to representing the lifelines and their communication. The set of lifelines is represented by a pattern layout. Its definition is shown in Figure 6.6. The simulation only displays squares as default visualization of the patterns. The representation of a lifeline uses a more complex background, as seen in Figure 6.5.

6.2.3.2. **Lifelines**. The projection for the lifeline uses a decoration-layout with a background imported from an SVG file. The name is directly placed in the middle of the top rectangle. The second rectangle is used to place the different elements attached to the lifeline. As the background may need to be adjusted to cover the different items of the set, a marker *data-augment* is defined in the vector to describe its evolution (**Req #3**). The creation of new elements is delegated to a button with a position that is adapted to represent the coordinates of the next item. It is placed directly at the bottom of the vector.

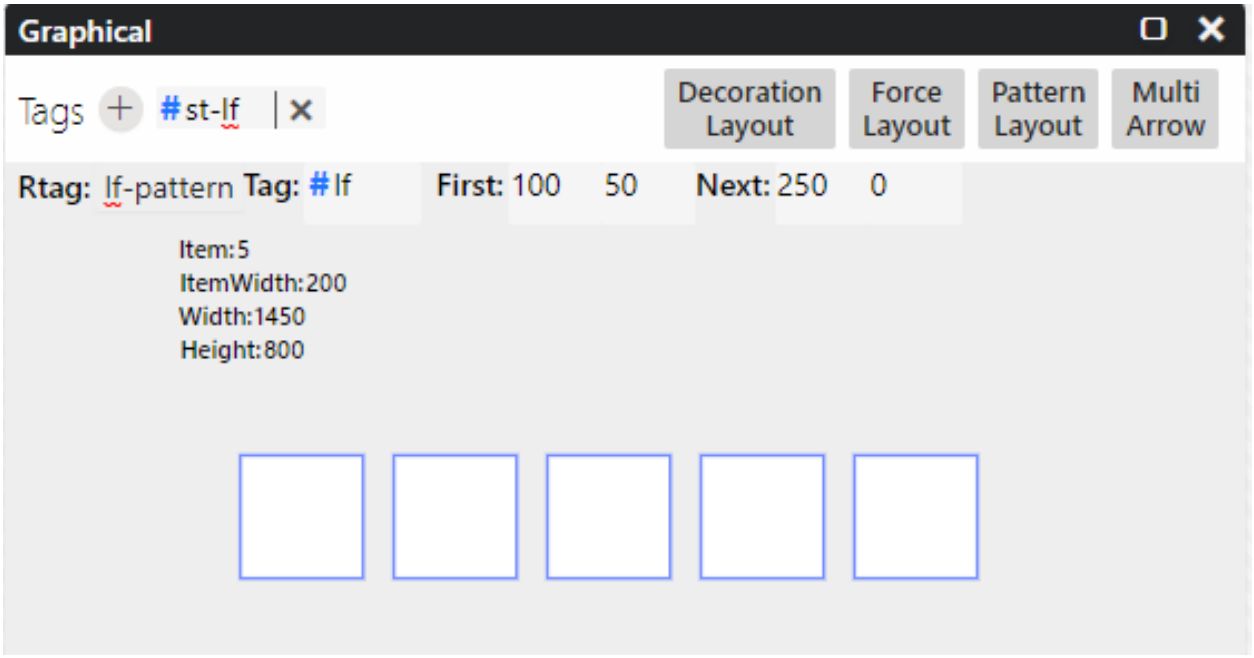


Figure 6.6. Pattern layout definition for the set of lifelines.

6.2.3.3. **Set of elements**. An *element* is a prototype extended by concepts with different semantics in the metamodel. The set of elements is defined with a pattern layout that contains information on the position of the first item on the lifeline and the operations that are applied to place the next element (**Req #2**). This choice was motivated by the idea that the space between two items is a constant only evolving on the vertical axis. In a sequence diagram, the different messages and responses are placed to express the general order of the sequence of instruction. To preserve this expressiveness in our editor, each set of elements is registered in an anchor template. This process allows the different concepts to have a common evolution when computing the position of their items.

Figure 6.7. Selection of the value of an element.

6.2.3.4. **Element**. Each element is represented with a black circle with a delete button in its center, as seen in Figure 6.7. The selection of the value is located on the left side of the projection with a choice-field (M for a Message, R for a *Response*, and L for a *Loop*).

A white button allows the end-user to open and close the selection. Each concrete concept may have a different representation based on its value. The projection offers an additional view of the concept value. If it has not been selected yet, nothing is rendered. The end-user may change the value of an element at will. All these projections are contained in a decoration-layout.

(a) Container-based projection of *Transition*.

(b) Communication between two lifelines.

Figure 6.8. Selection of the target attribute.

6.2.3.5. **Messages and responses**. Messages and responses use projection shadows to render a visual connector (**Req #6**), as seen in Figure 6.8(a). The interaction-based projection is a choice-field shown in Figure 6.8(b) that represents the different values for the target attribute. Each lifeline is represented with a text field that only shows the three first characters of its name to avoid managing the size of each cell. The edge projection is defined as an *Anchor-Edge*. It uses the anchor template to find the coordinates of the target lifeline in the editor. Since each anchor is registered in the template, the index of the source element guarantees an easy identification of the target position. The projection has a decorator describing its content and is placed using the minimum value on the x-axis between the source and target points.

6.2.3.6. **Fragments**. Loops require specific management. They can be placed on an anchor and contain elements that need to be located inside of their projection. To represent fragments, the projection uses a layout not presented in the previous chapter. The *Anchor-Layout*, shown in Figure 6.9, can be located at coordinates registered in an anchor template. When rendered, the layout creates a reservation. The reservation works as a sub-template with its indexes. Adding a new element uses the reservation and adapts the template to increase the coordinates of the elements with an index superior to the reservation. Anchors are not related to an attribute of the concept. After creating a message on a lifeline, it can be sent to an *anchor-layout* using its identifier. In our case, the identifier is defined as the order of a bloc. The layout then analyses the dimension of the object it receives to guarantee that

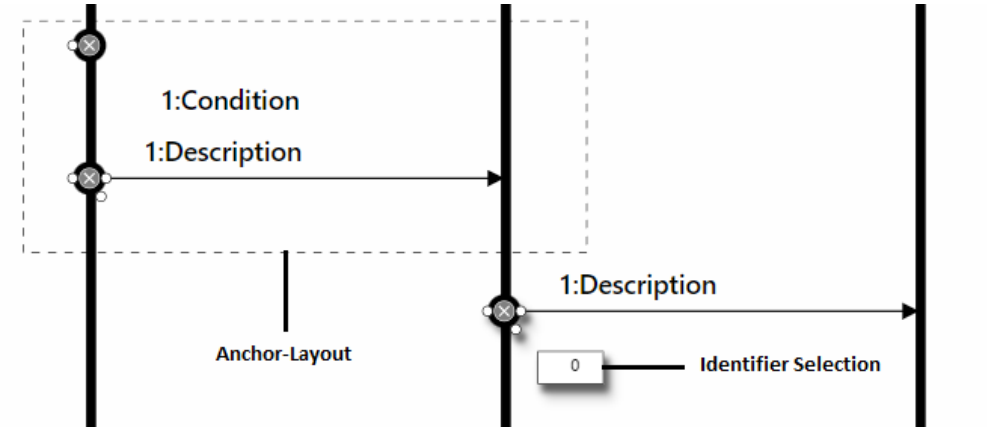


Figure 6.9. The *Anchor-Layout*.

it can cover it (**Req #3**). This layout is very context-specific and need to be generalized, that is the reason why we decided to not present it in Chapter 5.

6.2.4. Discussion

The sequence diagram editor offers great support for representing different scenarios. One main advantage of the projection model is that the interaction is concentrated in the graphics. No container-based projection shadow is created in the editor. One of the limitations is the creation of *Loops*. The *Anchor-Layout* is very specific to the context of a sequence diagram and requires modifications to make it more general and usable in other structures. As the layout still needs to be improved, we did not represent if/else blocks. Another concern is the management of the order in different sequences. The concept definition in Gentleman does not offer enough constraints on the set to order its elements yet. Moreover, interacting with the editor relies mostly on clicking on boxes. A drag-and-drop interaction in the editor would strongly improve the usability of this editor.

Requirement	**Management**
#1	This editor does not use any container-based projections.
#2	The lifelines and the elements are automatically placed.
#3	The dimension of each lifeline can adapt to the creation or suppression of an element.
#4	This requirement is respected.
#5	This requirement is respected.
#6	This requirement is respected.

Tableau 6.2. Requirements in the sequence diagram editor.

6.3. Music Sheet

Music notations allow their creators to visually represent the different notes and chords that compose a melody. Nowadays, musicians use a staff notation to explicitly describe the tempo and values of the different notes in a partition [**80**]. In software engineering, DSLs have been developed to help the creation of music such as pyTabs [**81**]. This solution focuses on the definition of a grammar to create music sequences with a tablature notation and to play them using a synthesizer. The disposition of the elements plays a crucial role in music notation. The value of a note can be directly deducted from its position on the stave. Creating a music editor allows us to explore the relation between the coordinates of a projection and its underlying concept. The projection model is available on github.[7]

6.3.1. Metamodel

```
1 concept root Sheet: {
2     attributes: [
3         { name: "name", type: string },
4         { name: "author", type: string },
5         { name: "staves", type: set, accept: Staff},
6         { name: "notes", type: set, accept: Note },
7         { name: "tuples", type: set, accept: Tuple }
8     ]
9 }
10
11 concept Staff: {
12     attributes:[]
13 }
14
15 prototype Note: {
16     attribute: [
17         { name: "tempo", type: string, accept: ["Black, "White"], default: "Black" }
18     ]
19 }
20
21 concept (Do, Re, Mi...): Note {}
```

Listing 6.3. Music Metamodel.

The root concept of the music metamodel shown in Listing 6.3 is the *Sheet*. A sheet has a name, an author, and different staves that hold the different notes of the melody. A *Note* is a prototype with twenty-four values to cover a staff. Since the position is related to the

[7] https://github.com/geodes-sms/gentleman/blob/master/models/music-model/projection.json

pitch of a note, each value has a dedicated concept. This binds the value to the coordinates in the projection. This process would not be possible in Gentleman if we used a number or a string. Each of them can have a tempo to express its duration. *Tuples* allow the subdivision of time. They are represented in the model with a set of references to existing notes. The metamodel does not represent chords since they require specific constraints not available yet in Gentleman.

6.3.2. Projections

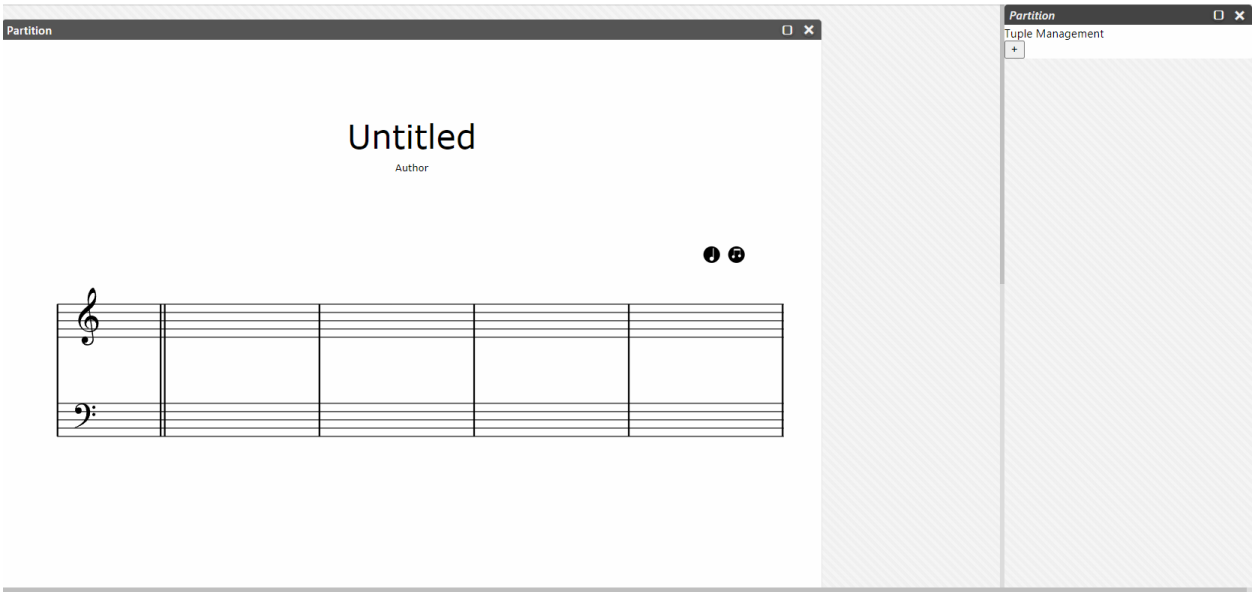


Figure 6.10. The music sheet.

6.3.2.1. **Sheet**. The music sheet has two projections in the model. A decoration-layout allows the interaction with the notes and a container-based projection deals with tuple management. The graphical projections use the benefits of the decoration-layout to simulate a document shown in Figure 6.10. The name of the song and its author are directly represented in the top part of the sheet. Two buttons allow the creation of new notes and the generation of the alternative view (**Req #4**). The body of the document contains the different staves where the notes are attached.

6.3.2.2. **Staves and set of notes**. In the metamodel, notes are not described as an attribute of the stave concept. Gentleman does not offer enough constraints to apply bounds to a set. The solution proposed is to use the concept of projection shadow to place directly the notes on the staves (**Req #2**). The first button shown in Figure 6.10 represents the set of notes in the model. This projection has a shadow that represents the notes in a pattern

layout, as the distance between two notes is fixed. As a receiver, the shadow points to a staff projection so that it can directly place the set representation in its SVG element. To limit the number of notes in a staff, a saturation is added to the pattern layout. Whenever the maximum capacity is reached, the projection asks the editor to create a new receiver. If a note is removed and the staff needed anymore, the concept is deleted. This communication is allowed because the set of staves is also represented by a pattern layout following a vertical flow.

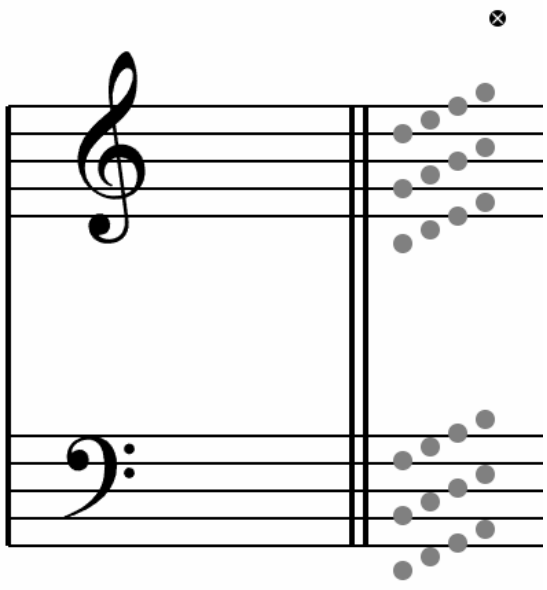


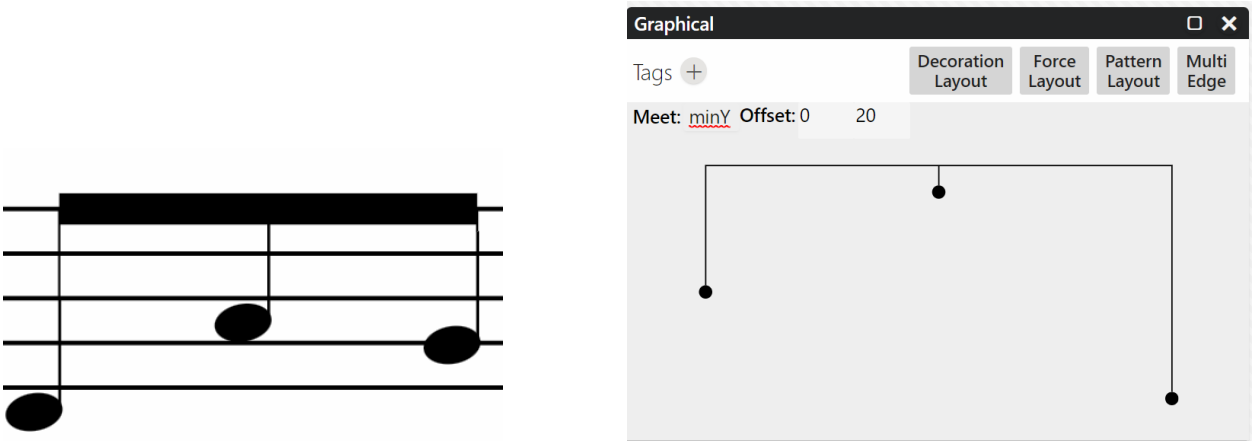
Figure 6.11. Selection of the value of a note.

6.3.2.3. **Note Selection**. A note can be defined by its position on the staff. To use this relation, the selection of a note uses a placeholder field (**Req #1**). Each note comes with two projections in the model. One is used as a placeholder with predefined coordinates as seen in Figure 6.11. The end-user only has to click on the placeholder to set the value for the concept. The second projection is then rendered to draw the note on the staff. Each value for a note has a specific background to ensure that the editor can represent the values that are not in the stave. A button is displayed in the top left corner of the projection to open the placeholder and modify the value if needed. This button is only shown when the SVG element is focused and the selection is opened.

6.3.2.4. **Tempo**. The tempo can have different values. As multiple buttons are already present in the projection, having an additional interaction box may hinder a good visualization of the concept (**Req #5**). To overcome this issue, each note has a switch field (see

Section 3.2.2) in its center to rotate through the different values of the concept. Clicking on a black note makes it white and conversely.

6.3.2.5. **Tuples**. Tuple management follows a process similar to the one defined in the Statecharts editor. The main difference is that a tuple is represented by a *Multi-Edge* shown in Figure 6.12. Every time the end-user adds a new reference to a concept, its value is analyzed to find the corresponding projection in the sheet. Connecting the different points then requires special treatment to respect music notations. In a *multi-edge*, constraints can be applied when rendering the projection by modifying an attribute called "meet". This attribute is represented as a string with five values: free, minX, minY, maxX, maxY. We are currently working on creating more complex values to draw splines. In our case, this attribute is set to minY. When the path is drawn, the edge looks in the list of points the one with the minimal value on the y-axis. Once this parameter has been computed, a straight line is created to cover all of the points on the x-axis (**Req #6**). This path is designated as the *main-path*. Style can be applied with CSS or SVG attributes. After creating the main path, a sub-path is added to connect the different points to it. They may have a dedicated style.



(a) A multi-edge representing a tuple.

(b) Definition of the *Multi-Edge*.

Figure 6.12. Projection for a tuple.

6.3.3. Discussion

The music sheet editor takes advantage of positions to maximize the interactions in the graphical projections. The presence of placeholders and the superposition of pattern layouts allow the end-user to focus on the meaning of note placement in the staff and not on the

adaptation of the concrete syntax. Even if the projections are robust enough, some improvement may be done on the metamodel. As constraints are very limited with Gentleman, staff and nodes are not related as concepts but they appear as connected in the concrete syntax. A second improvement would be the addition of chords to the editor. A solution might be to define a common prototype *Element* for chords and notes. However, a note would have to be defined as a concrete concept. Gentleman does not consider prototypes as potential values for an attribute. The end-user has to directly instantiate them with a concrete concept or a primitive. This implies that when the end-user creates a new *Element*, the editor considers that it can either be a chord or one of the twenty-four values for a *note*. This implies that the placeholder field could not be used here, as a chord is composed of multiple notes. To add chords in the metamodel, a strategy could be to create a method for grouping values by prototypes. This solution requires modifications of the concepts in Gentleman, while our extension only focuses on projections.

Requirement	**Management**
#1	The editor only uses container based projections for tuple-management.
#2	New notes are directly placed on a staff according to their value.
#3	No size management is required.
#4	The setting of the value of a note can only be realized with the placeholder field.
#5	This requirement is respected.
#6	This requirement is respected.

Tableau 6.3. Requirements in the sequence diagram editor.

6.4. Analysis

The three examples presented in this chapter cover different requirements that can be found in graphical editors. Statecharts allow us to focus on graph-based layouts, the complexity of the sequence diagrams shows how our different layouts can coexist in an editor, and finally, the music sheet raises the question of the relation between a concept and its placement. Overall, the results are satisfying. All of the editors are functional and can create syntactically correct models. The first conclusion is that the majority of the layouts presented during the thesis are mature enough to construct complex graphical structures. However, each application example can still be improved. After perfecting our layouts, the next step would be to compare our solution to other editors that use automatic-layout methods, like the KIEL STATECHART EXTENSION OF DOT [**36**].

Chapter 7

Conclusion

In this chapter, we conclude the thesis. We start by briefly discussing the different points we presented before talking about the future works.

7.1. Summary

Projectional editors focus on structure-based editing to offer freedom of representation for the concepts of a model. Solutions like MPS [**10**] and Gentleman [**41**] have proven their utility to create text-based and container-based projections, and to integrate the resulting editors in various applications. However, these frameworks lack support for graphical notations, even though structure-based editing has been a much-needed requirement for graphical languages [**35**]. A formal definition of projectional editing for these languages still has not been made yet.

In this thesis, we implemented an extension of Gentleman for graphical projections. Using SVG, we created multiple interaction-based structures and layouts to organize them. From the extension, we extracted requirements that are inherent to graphical and projectional editing. First, the editor should automatically manage the disposition and size of the elements. This ensures that the end-user only interacts with the model and does not have to take time-consuming activities like positioning the elements into consideration. Since a model contains information transmitted with the notations, automatically managing size and position requires that visual aesthetics are taken into account to facilitate the visualization of the underlying structures. Following the same principle, each interaction point should have its own projection to ensure that the end-user can easily predict the actions related to specific parts of complex graphics. Connecting the different elements can become a challenge as the interaction in projectional editing still needs to be improved [**9**], so the entire context of an edge projection should be resolved before rendering. This prevents the presence of any syntactically incorrect structures in the projections. Finally, the interaction should be kept in the graphics to make the difference between projectional editors and visualization tools.

Based on these requirements, we proposed different approaches that can be adopted in the resulting editors. As layout management plays a crucial role in graphical projections, we presented methods to implement graph-based layouts and other specific structures by identifying the concepts they could be related to and the different behaviors and parameters they require. These different layouts are available in our extension and were used to generate three application examples. These editors cover basic interaction and can still be improved, but they show promising results.

7.2. Future Work

One of the crucial points of this thesis was the creation of different layouts to manage the automatic positioning of the elements. We proposed simple solutions that cover basic representations that can be found in various contexts. However, some additional structures could be imagined. For example, the tree layout presented in Chapter 5 only considered a hierarchical view of a tree. This layout could not be used in the context of the family tree discussed in Chapter 3. Having more layouts with additional parameters or considering an approach similar to the one available with KIELER [**34**] could offer more customization for the language engineer when creating new projections. Currently, we are working on improving the *Anchor-Layout* briefly discussed in Chapter 6 and implementing a *Grid-Layout* in Gentleman. The latter could offer an alternative to the *Force-Layout* to generate a graph-based representation of a model as it takes inspiration from orthogonal drawing techniques.

This thesis strongly focused on the definition and management of graphical projections. Because projectional editing focuses on interacting with the abstract syntax, the next step is to consider the question of navigation and a new type of interaction. We explained in Chapter 4 issues of the click-and-drag interaction in a projectional editor. However, this type of interaction should be explored to diversify the user experience. The current editing activities mostly rely on textual inputs and mouse clicks. Focusing on navigation between the different elements of a layout and having a formalism to define interactions specific to a projection could strongly improve the way the end-user creates and modifies the ASG while staying in the graphics.

After working on the navigation, an idea would be to work on the addition of multiple view for a concept. For example, in our statechart editor, the end-user could select a subregion of a composite and zoom in the projection to see the states it contains and how they interact. Similarly, she could select filters in a panel to only see a specific type of projections for the different instances of a concept. This approach would be very interesting to visualize the components of the model and offer multiple ways to interact with their attributes. To implement this idea, the first step would be to work on the concept of properties (e.g., the

number of elements in a collection) that already exists in Gentleman in order to be able to represent them in the projections.

Finally, our last objective is related to the extension on Gentleman. As explained in Chapter 3, container-based projections cannot be embedded in the graphics yet. The two projection models are separated which can be confusing for the language engineer. As a short-term solution, we are trying to merge the two models while creating constraints on the embedding possibilities. In the long run, we are imagining a special structure of listeners and handlers to automatically adapt the SVG elements when an HTML container changes its size. This will allow the two types of projections to coexist in a single representation of a concept.

References

[1] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. In *International Conference on Model Driven Engineering Languages and Systems*, pages 98–110. Springer, 2006.

[2] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[3] Farah Arefi, Charles E. Hughes, and David A. Workman. Automatically generating visual syntax-directed editors. *Communications of the ACM*, 33(3):349–360, 1990.

[4] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *IEEE software*, 31(3):79–85, 2013.

[5] Benoît Langlois, Consuela-Elena Jitia, and Eric Jouenne. DSL classification. In *OOPSLA 7th workshop on domain specific modeling*, 2007.

[6] Martin Fowler. Language workbenches: The killer-app for domain specific languages, 2005.

[7] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+ defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93, 2003.

[8] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. Atompm: A web-based modeling environment. In *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*, pages 21–25, 2013.

[9] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. Efficiency of projectional editing: A controlled experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 763–774, 2016.

[10] Fabien Campagne. *The MPS language workbench: volume I*. 2014.

[11] Markus Voelter and Sascha Lisson. Supporting diverse notations in mps'projectional editor. In *GEMOC@ MoDELS*, pages 7–16, 2014.

[12] Bran Selic. A systematic approach to domain-specific language design using uml. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 2–9. IEEE, 2007.

[13] Dániel Varró. Model transformation by example. In *International Conference on Model Driven Engineering Languages and Systems*, pages 410–424. Springer, 2006.

[14] Ivar Jacobson, Grady Booch, and James Rumbaugh. The unified modeling language. *University Video Communications*, 1996.

[15] A Roques. PlantUML: Open-source tool that uses simple textual descriptions to draw UML diagrams, 2015.

[16] Frédéric Jouault and Jean Bézivin. Km3: a DSL for metamodel specification. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

[17] Dimitrios S Kolovos, Antonio García-Domínguez, Louis M Rose, and Richard F Paige. Eugenia: towards disciplined and automated development of GMF-based graphical model editors. *Software & Systems Modeling*, 16(1):229–255, 2017.

[18] Luis Mandel and Maria Victoria Cengarle. On the expressive power of OCL. In *International Symposium on Formal Methods*, pages 854–874. Springer, 1999.

[19] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016.

[20] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. Tcs: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, 2006.

[21] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of concrete textual syntax mapping approaches. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 169–184. Springer, 2008.

[22] Blazo Nastov and François Pfister. Experimentation of a graphical concrete syntax generator for domain specific modeling languages. In *INFORSID*, pages 197–213, 2014.

[23] Raphaël Mannadiar and Simon Van Mierlo. Atompm user's manual. 2016.

[24] Christian Brand, Matthias Gorning, Tim Kaiser, Jürgen Pasch, and Michael Wenz. Development of high-quality graphical model editors. *Eclipse Magazine*, 1, 2011.

[25] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[26] Markus Scheidgen. Textual modelling embedded into graphical modelling. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 153–168. Springer, 2008.

[27] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of DSM graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.

[28] Graphical Modeling Framework. GMF, 2015.

[29] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In *International Conference on Model Driven Engineering Languages and Systems*, pages 196–210. Springer, 2010.

[30] Denis Dubé. Graph layout for domain-specific modeling. M.Sc. thesis, McGill University, 2006.

[31] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. In *International Symposium on Graph Drawing*, pages 155–166. Springer, 2004.

[32] Achilleas Papakostas and Ioannis G Tollis. A pairing technique for area-efficient orthogonal drawings. In *International Symposium on Graph Drawing*, pages 355–370. Springer, 1996.

[33] Emden R Gansner and Stephen C North. Improved force-directed layouts. In *International Symposium on Graph Drawing*, pages 364–373. Springer, 1998.

[34] Miro Spönemann, Christoph Daniel Schulze, Christian Motika, Christian Schneider, and Reinhard von Hanxleden. Kieler: Building on automatic layout for pragmatics-aware modeling. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 195–196. IEEE, 2013.

[35] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. In *Monterey Workshop*, pages 116–140. Springer, 2008.

[36] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *International Conference on Model Driven Engineering Languages and Systems*, pages 635–649. Springer, 2007.

[37] Bence Graics and Vince Molnár. Formal compositional semantics for Yakindu Statecharts. In *24th PhD Mini-Symposium (Minisy@ DMIS 2017)*, pages 22–24. Budapest University of Technology and Economics, 2017.

[38] Raul Medina-Mora and Peter H. Feiler. An incremental programming environment. *IEEE Transactions on Software Engineering*, (5):472–482, 1981.

[39] David Notkin. The gandalf project. *Journal of Systems and Software*, 5(2):91–105, 1985.

[40] Thomas Reps and Tim Teitelbaum. The synthesizer generator. *ACM Sigplan Notices*, 19(5):42–48, 1984.

[41] Louis-Edouard Lafontant and Eugene Syriani. Gentleman: a light-weight web-based projectional editor generator. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, pages 1–5, 2020.

[42] Riccardo Solmi. *Whole platform*. PhD thesis, Citeseer, 2005.

[43] Markus Voelter, Daniel Ratiu, Bernhard Schaetz, and Bernd Kolb. mbeddr: an extensible c-based programming language and ide for embedded systems. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 121–140, 2012.

[44] Tomáš Fechtner. Mps-based domain-specific language for defining rtsj systems. 2012.

[45] Vaclav Pech, Alex Shatalin, and Markus Voelter. Jetbrains mps as a tool for extending java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 165–168, 2013.

[46] Louis-Edouard Lafontant. Generating web-based projectional editors. M.Sc. thesis, Université de Montréal, sep 2021.

[47] Brice Michel Bigendako. Relis: a flexible tool for conducting systematic reviews iteratively and collaboratively. M.Sc. thesis, Université de Montréal, 2018.

[48] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*, pages 41–61. Springer, 2014.

[49] Kenneth R Sloan and Steven L Tanimoto. Progressive refinement of raster images. *IEEE Transactions on Computers*, 28(11):871–874, 1979.

[50] Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. *Scalable vector graphics (SVG) 1.0 specification*. iuniverse Bloomington, 2000.

[51] Frédéric Fondement. Graphical concrete syntax rendering with svg. In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 200–214. Springer, 2008.

[52] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52:43–62, 2018.

[53] Gerhard Viehstaedt and Mark Minas. Interaction in really graphical user interfaces. In *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 270–277. IEEE, 1994.

[54] I Scott MacKenzie and William Buxton. Extending Fitts' law to two-dimensional tasks. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 219–226, 1992.

[55] Jeff Johnson. *Designing with the mind in mind: simple guide to understanding user interface design guidelines*. Morgan Kaufmann, 2020.

[56] I Scott MacKenzie and Kumiko Tanaka-Ishii. *Text entry systems: Mobility, accessibility, universality.* Elsevier, 2010.

[57] Kirstin Krauss. Visual aesthetics and its effect on communication intent: a theoretical study and website evaluation. *Alternation*, 12(1a):305–329, 2005.

[58] Johannes Pölz. *UML diagram and element generation exemplary study on UMLet.* PhD thesis, 2009.

[59] Daniel Moody. The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering*, 35(6):756–779, 2009.

[60] Nick Cawthon and Andrew Vande Moere. The effect of aesthetic on the usability of data visualization. In *2007 11th International Conference Information Visualization (IV '07)*, pages 637–648, 2007.

[61] Talia Lavie and Noam Tractinsky. Assessing dimensions of perceived visual aesthetics of web sites. *International journal of human-computer studies*, 60(3):269–298, 2004.

[62] Markus Voelter, Bernd Kolb, Tamás Szabó, Daniel Ratiu, and Arie van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with mps. *Software & Systems Modeling*, 18(1):585–630, 2019.

[63] Andreas Noack. Modularity clustering is force-directed layout. *Physical Review E*, 79(2):026102, 2009.

[64] Arjun Srinivasan, Hyunwoo Park, Alex Endert, and Rahul C Basole. Graphiti: Interactive specification of attribute-based edges for network modeling and visualization. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):226–235, 2017.

[65] Henry Heberle, Marcelo Falsarella Carazzolle, Guilherme P Telles, Gabriela Vaz Meirelles, and Rosane Minghim. Cellnetvis: a web tool for visualization of biological networks using force-directed layout constrained by cellular components. *BMC bioinformatics*, 18(10):25–37, 2017.

[66] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, 1995.

[67] Phillip Whitt. An overview of Paint.NET. *Practical Paint.NET*, pages 1–38, 2022.

[68] Helen C. Purchase. Effective information visualisation: a study of graph drawing aesthetics and algorithms. *Interacting with computers*, 13(2):147–162, 2000.

[69] Michael Jünger and Petra Mutzel. Technical foundations. In *Graph Drawing Software*, pages 9–53. Springer, 2004.

[70] Pablo Navarro Castillo. *Mastering D3. js.* Packt Publishing Ltd, 2014.

[71] Peter Eades. *Drawing free trees.* International Institute for Advanced Study of Social Information Science, 1991.

[72] Chun-Cheng Lin, Hsu-Chun Yen, et al. On balloon drawings of rooted trees. *J. Graph Algorithms Appl.*, 11(2):431–452, 2007.

[73] John Q Walker. A node-positioning algorithm for general trees. *Software: Practice and Experience*, 20(7):685–705, 1990.

[74] Roberto Tamassia. *Handbook of graph drawing and visualization.* CRC press, 2013.

[75] Michael Von der Beeck. A comparison of statecharts variants. In *Formal techniques in real-time and fault-tolerant systems*, pages 128–148. Citeseer, 1994.

[76] Andreas Muelder. Yakindu. *Yakindu Statechart Modeling Tools*, 2011.

[77] Jim Barnett. Introduction to SCXML. In *Multimodal Interaction with W3C Standards*, pages 81–107. Springer, 2017.

[78] Xiaoshan Li, Zhiming Liu, and He Jifeng. A formal semantics of uml sequence diagram. In *2004 Australian Software Engineering Conference. Proceedings*, pages 168–177. IEEE, 2004.

[79] Oksana Ņikiforova, Sergii Putintsev, and Dace Ahiļčenoka. Analysis of sequence diagram layout in advanced uml modelling tools. *Applied Computer Systems*, 19(1):37–43, 2016.

[80] Mark McGrain. *Music notation*. Hal Leonard Corporation, 1990.

[81] Miloš Simić, Željko Bal, Renata Vaderna, and Igor Dejanović. Pytabs: A DSL for simplified music notation. In *The 5th International Conference on Information Society and Technology (ICIST 2015), edited by M. Zdravković, Trajanović, M., Konjović*, number 439-43, 2015.