SN Computer Science (2023) 4:189
https://doi.org/10.1007/s42979-022-01574-3

ORIGINAL RESEARCH

Classification for the Concrete Syntax of Graph-Like Modeling Languages

Gregor Wrobel[1] · Robert Scheffler[1]

Received: 12 June 2022 / Accepted: 16 December 2022 / Published online: 2 February 2023

Abstract
A classification scheme for Graph-Like Modeling Languages (GLML) is presented in this paper. The novelty of this classifier lies in its application to a meta-model for GLML that deviates from the simple graph model and underlies a large number of GLML. The main goal of using this classification scheme is to support the reuse of layout algorithms for GLML. GLML are used directly or indirectly for the development of software by model-based software engineering techniques. In other domains, graph-like models are artifacts (e.g., circuit diagrams, energy flow diagrams) that serve as input for downstream specialized applications (simulators, optimizers). The concrete syntax of a language for creating, editing, and understanding models is highly important for the development of modeling tools. Layout methods for the used languages have to be implemented to achieve software tools with good usability. Developing layout algorithms is a complex topic that is covered by the specialized field of Graph Drawing. However, there is no existing procedure to determine which layout algorithm can be used for a GLML. Matching layout algorithms to GLML can be achieved by applying the presented classification scheme.

Keywords Concrete syntax of modeling languages · Graph-like modeling languages · Graph drawing · Layout algorithm · Domain-specific modeling

Introduction

Abstract models and modeling languages are used in software engineering and classical engineering sciences to describe systems. Graph-like languages were introduced as suitable modeling tools as early as the turn of the twentieth century. Many inventions in the field of electricity were published in patent specifications using graph-like visualizations (e.g., [1]). The prevalence of computer technology and especially the propagation of model-based design (MBD) have increasingly led to the rise of GLML in the sciences.

This article is part of the topical collection "Advances on Model-Driven Engineering and Software Development" guest edited by Luís Ferreira Pires and Slimane Hammoudi.

✉ Gregor Wrobel
wrobel@gfai.de

Robert Scheffler
scheffler@gfai.de

[1] Graph Based Engineering Systems, Society for the Advancement of Applied Computer Science, Volmerstraße 3, Berlin, Germany

The concrete syntax of a language is of utmost importance for the understanding of the language [2] and the main means by which users interface with models [3]. In this context, the comprehensibility of modeling languages strongly depends on the modeling skills of the users [4]. The users deploying model-based software development (MBSD) are typically software engineers with experience in abstract languages (e.g., programming languages) and general-purpose modeling languages (GPML) like UML. In model-based engineering (MBE) the users are often classical engineers with little modeling experience. They are supported by domain-specific languages (DSL).

Another important aspect is the usage of the created models. Applications that only need a single model to be generated do not have high demands on the usability of the modeling process. But when models are created and edited frequently, the modeling itself becomes an important part of the user's work. This is then linked to high demands regarding usability, comparable to the demands of UI/UX design.

To meet these requirements, modeling tools have to offer algorithms both for drawing of and interacting with the GLML.

The layout of GLML is complex compared to the concrete syntax of textual languages. Although graph drawing is the specialized field that is concerned with the visualization of graphs, it has not systematically been applied in MBD yet [5]. There are two reasons for this. On one hand, the automatic drawing of graphs is less important than the implementation of layout algorithms and interaction methods that support the creating and editing of models. Here aspects such as dynamic graph drawing and layout stability are more important than static graph drawing. On the other hand, graphical models within the scope of MBD are structurally very varied. They differ from the classic, simple graph model consisting of vertices and edges. Graph models in MBD can be port graphs, hypergraphs, nested graphs, and labeled graphs. In this paper, these models are encapsulated in the term graph-like.

The multitude of different graph models, which are often defined in the concrete syntax of the metamodels for GLML, and the different use cases for layout algorithms complicate their reuse and adaptation considerably. Reuse and compatibility (of languages) are two of the TOP 10 challenges faced in MDE artifact sharing [6]. Typical artifacts in MDE include models, metamodels, model transformations, and modeling tools. In addition, for widely used languages, especially GPML, the concrete syntax is defined only very superficially, even though it should be an important part of the language. For UML 2.5.1, the specification takes up just 20 of 754 pages of documentation [7]. In SysML 1.6 [8], the concrete syntax describes only the graphical aspects of the language.

A major shortcoming is that for many languages there is de facto no sufficient definition of the concrete syntax to provide state-of-the-art layout algorithms.

This paper presents a classification scheme for the concrete syntax of GLML. This provides the possibility to assign layout algorithms not to a specific language or tool, but to a class of modeling languages according to the classification scheme. A classifier provides developers with the ability to more effectively use existing layout algorithms, which may be grouped in libraries (e.g., the Eclipse Layout Kernel [9]), for the layout of GLML. In particular, if layout algorithms can be used via parameters for the layout of different concrete syntaxes, classifiers enable the mapping between GLML and layout algorithms. This facilitates the reuse and adaptation of layout algorithms and greatly simplifies tool development.

The classification scheme as presented contains structural, geometrical, and topological information. It explicitly excludes information on the graphical design of the GLML elements.

The proposed classification scheme is an expansion of [10] based on a different notation and was applied for a mapping between GLML and layout algorithms in [11].

“Related Works” describes the related works. “Features and Metamodel of GLML” presents a metamodel for GLML. The metamodel describes the essential model elements, for which features are listed in the classification scheme. “Layout Aspects of GLML” characterizes layout algorithms for GLML and differentiates between static graph drawing and dynamic graph drawing. In “Classification Scheme”, the classification scheme is detailed. Using the examples of the classical graph model, a GPML, and a DSL, “Examples” applies the classification scheme. “Layout Algorithm Reuse” shows a way to apply the classification scheme for layout reuse. “Conclusion and Further Work” gives a conclusion and outlook on further work.

Related Works

The concrete syntax of a GLML is closely related to the layout of a language also often referred to as visual notation. The following is a literature review of the related fields and publications.

User Experience and Aesthetic Aspects of GLML

The importance of aesthetic aspects in graphical languages is highlighted in many papers. Graphical languages are the interface between the user and the model of a specific domain.

Reference to individual language elements and their concrete syntax occurs repeatedly in the studied aspects concerning the aesthetics of a language. In [12], six different basic principles that improve the quality of information models are presented. Within the principle of clarity, the layout design is explicitly named as an essential factor; and the reference to aspects of aesthetics in graph drawing related to the edge geometry is established in [13].

Extensive research on the importance of aesthetics has been conducted by Purchase. Especially line crossings and line bends [14–19], and also, but less significant, line orthogonality and line angles [16, 19] are crucial properties of the layout of a diagram, which are directly related to the concrete syntax.

In [4], the positions of language elements of domain-specific graphical languages are highlighted as an important aspect for coping with complex modeling scenarios. As an example, the positions of ports for incoming (left) and outgoing (right) connections are given. Position properties (especially of ports) are part of the concrete syntax of a language.

Recent studies apply machine learning methods to evaluate the layout quality of diagrams [19] or refer to aspects of aesthetics for special domains, e.g., business process modeling [20, 21].

Unlike the extensive earlier work, this paper is not concerned with examining aspects of the aesthetics of the layout of GLML. Rather, the presented scheme introduces an approach not to limit the aesthetic aspects to a specific language or a few parameters of language elements (e.g., length, bends, and crossings of lines), but to relate them to the concrete syntax of languages. Investigations on the aesthetics or usability of languages that refer to the concrete syntax and already existing study results that can be transferred to a concrete syntax can then be used for a whole class of GLML.

Graphs and Graph Drawing

A great number of works on graph drawing originated in the 1990s. As an example, we mention the works of Di Battista et al. [22, 23]. Graph drawing methods exist for different classes of graphs. In [23], a general framework for graph drawing is presented, which contains parts of the features of the presented classification scheme (e.g., edge direction and routing classification). That framework, however, is strongly focused on concrete layout aspects (planarity) and properties of graphs (connectivity). Ports, nested graphs, hypergraphs, and labeling are not mentioned in this framework. In [22], algorithms for drawing graphs are classified (into the classes trees, general graphs, planar graphs, and directed graphs), and a literature study on this differentiation is conducted. In addition, a few structural feature distinctions were considered (hypergraphs, compound graphs). There are other classification approaches for other special properties of graphs. For port graphs, there is a classification regarding an important feature, the port position, in connection with the development of specific layout methods [24].

Problems of labeling were researched intensively in the environment of geographic maps. In [25], a model with 9 different possibilities for the placement of axis-parallel enveloping label rectangles is presented. An overview of publications on map labeling is provided by [26].

Label orientation is also a relevant and widely researched topic in traffic maps. An overview of existing work is provided by [27].

The vast majority of this work is based on graphs as a metamodel, classically consisting of vertices and edges. Besides, there are languages in which the connection points (ports) are anchored in the metamodel as elements in addition to the vertices and edges.

Netlike Schematics

In [28], netlike schematics is the term used to differentiate from graphs. Graphs are considered besides electrical diagrams, technological layouts, and petri nets as examples of netlike schematics. A general language called schematic structure description language (SSDL) is presented. The essential difference of the examined languages to graphs is that instead of nodes, components connected by ports are part of the languages. Furthermore, the concrete relation to industrial applications (functional block diagrams, logic diagrams, flowcharts, and circuit schematics) was given. The work in [28] originated at the Central Institute of Cybernetics and Information Processes, Academy of Science of the G.D.R. A whole series of works on automatic layout synthesis of netlike schematics was carried out there in the late 1980s [29–32].

The work on this has been continued in the following years, and the results have been integrated into a software framework [33]. The connection to industrial applications was maintained, and a number of domain-specific languages [34–36] with integrated layout methods [37, 38] based on this framework have been developed.

Additional metamodels corresponding to these netlike schematics are currently integrated into some frameworks [9, 33, 39, 40] and implemented in layout algorithms. But also in these frameworks, not every layout algorithm can be used for every concrete syntax based on the metamodel, and there are no mapping mechanisms to get information about suitable layout methods already during language engineering.

Technical Languages

Modeling languages with a precisely defined concrete syntax exist, especially in engineering domains as technical languages. A prominent example of this are circuit diagrams. Before the advent of computer technology, circuit diagrams were drawn by hand. Today, extensive software tools exist for this purpose under the name electronic computer-aided design (ECAD). An advantage of the representation of electrical circuits is that there are standards that have existed for a very long time and are accordingly well evaluated. These standards contain concrete syntax specifications as well as aspects of drawing: “Lines between symbols should be horizontal or vertical with a minimum of line crossings, and with spacing to avoid crowding” [41].

Other technical languages that define their concrete syntax are IDEF [42, 43] and function block diagrams in IEC 61131-3 [44].

Languages in Model-Based Engineering

The specification of GLML is provided in different ways using different methods [45]. The languages in model-based engineering are typically defined by a meta-model [46]. Meta-models capture the relevant domain abstractions and concepts by specifying the abstract syntax and static semantics (aka well-formedness rules) of a graphical modeling

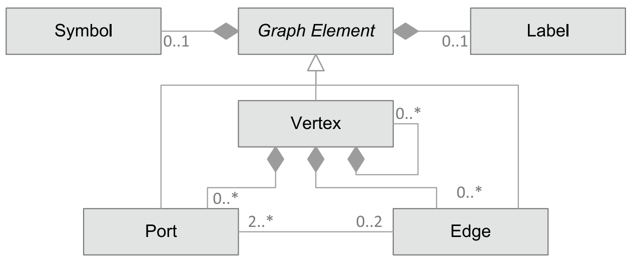


Fig. 1 A meta-model of GLML

language, whereas concrete syntax has received less attention. This stands in contrast to the design of domain-specific languages, where concrete syntax is an important design aspect [2]. This observation applies to both the current state of practice and the academic literature. For example, the latest UML specification (version 2.5.1) devised by the OMG [7] comprises a total amount of 754 pages, 678 of which are dedicated to the definition of the language's abstract syntax. UML diagrams are only considered in the Appendix, with the primary goal of standardizing the exchange of UML diagrams across tool boundaries. The actual concrete syntax is given in the form of a table (from page 704 to 723) which maps conceptual model elements defined by the abstract syntax to their graphical notation (i.e., shapes). The same observation applies to GPML which have been proposed in the academic literature, where the focus is typically on defining the abstract syntax as well as static and dynamic semantics, whereas the concrete syntax is, if at all, illustrated by a few examples (see, e.g., [47–49]).

Originating from model-driven development (MDD), so-called language workbenches are also to be considered. In language workbenches, the abstract and the concrete syntax as well as the mapping between them is defined as a meta-model [50–54, 54], and graphical editors for DSL are generated from it. Here are the challenges in developing suitable layout algorithms, which are currently not always well-solved for practical applications [55].

The presented classification scheme distinguishes vertices, ports, edges, labels, and symbols as the most important model elements. These model elements are explicitly included in several meta-models [9, 28, 33, 39], but the authors are not aware of any classification scheme for the concrete syntax of GLML.

Features and Metamodel of GLML

Figure 1 shows a possible meta-model for GLML. The essential model elements are vertices, ports, edges, labels, and symbols. Vertices, ports, and edges have a symbol that represents their graphical expression and reflects their domain-specific semantics. The symbols, as long as their shape is static, are irrelevant for layout algorithms and are not included in the classification scheme. They are used in the visualization to have distinguishable and recognizable model elements.

GLML are not "classical", simple graphs. The main difference is that the edges do not directly connect vertices, instead the connection is facilitated via ports that belong to the vertices.[1]

Ports for connecting vertices are used in many graphical languages. In some of those languages, ports are explicitly specified, and they also have a graphical or textual form, for example with fixed ports in ladder diagrams and function block diagrams in [44]. In other languages, ports function as positions for the connection of edges and vertices. These ports can have no expression and would then not be recognized as such in a graphical representation.

Some graphical modeling languages implement nested vertices. These contain other vertices and edges and can also be connected like non-nested vertices via edges (e.g., UML activity diagrams).

An important aspect of the concrete syntax of graphical languages is the specific graphic representation of language elements. Visual properties like color, line style, arrows, etc. make elements of graphical languages distinct and can emphasize their domain-specific context. A detailed study on the visualization of uncertainty in data using visual properties of edges (lightness, grain, fuzziness, transparency, width, hue, and saturation) shows the influence of these properties on the user [56]. Layout algorithms can abstract from the specific graphical properties. Because a detailed specification of shape features is not relevant, this paper omits it.

Layout Aspects of GLML

A separate field, graph drawing, exists for the development of layout methods for graph-like languages. Many fundamental graph drawing algorithms were developed and studied in the past decades [23]. Graph drawing algorithms are generally divided into procedures for placing vertices and procedures for routing edges. Many graph drawing methods solve both tasks together. Developed for drawing (simple) graphs consisting of vertices and edges, graph drawing methods are also used for the layout of GLML. However, GLML have special properties different from simple graphs, which have to be considered for the layout of the language. Vertices have dimensions as well as ports with graphical

[1] In the metamodel in Fig. 1, edges must always be connected to ports. To use this metamodel to represent a graph model without ports, each vertex is assigned exactly one port that has no symbol and is located on the vertex.

expressions and defined positions. This makes the development of layout algorithms a more challenging task than simply adapting graph drawing algorithms.

Graph drawing methods can be roughly divided into static graph drawing and dynamic graph drawing. In static graph drawing, an existing set of vertices and edges is laid out. dynamic graph drawing, on the other hand, deals with the layout of entire sequences of graphs [57]. For example, a layout can be transformed into another layout by deleting/adding edges/vertices, changing edges, moving vertices (including the following collision handling), and expanding vertices.

For the GLML characterized in the previous chapter, dynamic graph drawing is of greater importance than static graph drawing, especially when GLML are artifacts of modeling tools and/or editors. In this environment, the user is faced with the task of creating a new model (on a "blank sheet of paper") or modifying a previously created model. To cover these use cases well, it is helpful for dynamic graph drawing to preserve the mental map of the user [58]. The concept of preserving the mental map is a layout aspect of graph drawing that became the focus of research in the software engineering environment and there through the increasing prevalence of graphical modeling tools [59].

Besides the preservation of the mental map, aesthetic criteria for the layout of GLML are of importance. In addition to general design principles for graph drawing, the so-called Gestalt laws [60], and general design techniques in the context of graph drawing [61], specific criteria and metrics for the aesthetic of drawn graphs were developed [62]. In addition, domain-specific context (e.g., for UML diagrams [18]) or structural circumstances (e.g., for models of real technical systems [63]) is important for layout. The preservation of the mental map as well as general and domain-specific aesthetic aspects compete in the development of layout algorithms for GLML [16, 64].

The above-mentioned aspects of the layout algorithms for GLML complicate their development.

Classification Scheme

"Classification Scheme" provides a classification scheme for GLML. The first subchapter introduces the notation of the scheme, and the following subchapters describe the classifiers for vertices, ports, and edges of GLML.

Notation

The notation for the classification scheme is based on feature diagrams. Feature diagrams were introduced as part of the so-called Feature-Oriented Domain Analysis (FODA) [65]. Their notation was expanded in following publications. The notation in this paper uses the expansions by [66–69].

The syntax of feature models was extended to shorten the classifier and to designate properties as default when a feature is not specified by a GLML. These extensions can be found in Table 1 in rows 3 and 5.

The syntax in Table 1 is used in this paper.

GLML Classification

According to the meta-model in Fig. 1, a GLML has vertices, edges, and ports.

Port classification is optional. If no ports are classified for a given GLML, the GLML implements the classical graph model consisting of vertices and edges (Fig. 2).

The next chapters present the classification scheme for GLML elements. The simple GLML examples in Fig. 3 are used to illustrate the classifiers.

Common Classsification

The three GLML elements have a common set of abstract properties in the *Label classifier*.

Label Classification

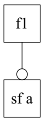
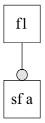
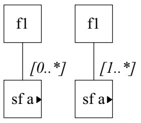
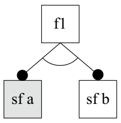
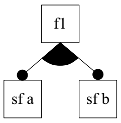
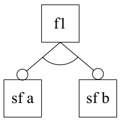
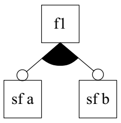
Vertices, edges, and ports can have labels. Labels describe the other elements in more detail and make them distinguishable and recognizable. Ports must be distinguished from each other, for example, to ensure that they are connected to valid edges (e.g., in circuit diagrams according to [70]). Vertex and edge labels are the subjects of specific research within the discipline of graph drawing. The first papers on this topic were published on the domain of cartography [71]. Vertex labels without overlap and label alignment are important for transit maps, and label placement is a central focus of studies next to schematic layout concerns [27].

The label classifier has a default value of *no label*. If a GLML element has a label, its position can be either fixed (*fix*) or freely placeable (*free*). The classification scheme provides the optional definition of the rotation properties, which allow possible rotations either in 90° steps or freely. The examples A, C, and D in Fig. 3 have labels. Example C allows a rotation of the port label (Fig. 4).

Vertex Classification

The main classification features for vertices regarding the graph type are the existence of ports (port graph), nesting (nested graph or not), and label (labeled graph or not). The *placement* classifier provides information about the desired

Table 1 Notation

Symbol	Explanation
m and *f*	A feature can take one of the values *m* (mandatory) and *f* (forbidden)
	Feature *f1* with optional solitary subfeature *sf a*
	Feature *f1* with solitary subfeature *sf a*. If the subfeature *sf a* is not set the default value of *sf a* will be set. Subfeauture *sf a* must have a default value
	Feature *f1* with mandatory (*[1..*]*) or optional (*[0..*]*) subfeatures *sf a* or *sf a*
	Feature *f1* with exactly one subfeature from a group. If the subfeature is not present, the default value (grayed out) will be set and *f1* can be omitted (see above). The specific subfeature that is set has the value *m* (mandatory). All other subfeatures have the value *f* (forbidden)
	Feature *f1* with at least one subfeature of a group. The set subfeatures have the value *m* (mandatory), all other subfeatures have the value *f* (forbidden). If more than one subfeature has the value *m*, it means that one of these subfeatures can be selected. The value *m* means optional in this case
	Feature *f1* with exactly one subfeature from a group that is optional. The set subfeature has the value *m* (mandatory). All other subfeatures have the value *f* (forbidden)
	Feature *f1* with optionally one or more subfeatures of a group. The subfeatures have the value *m* (mandatory) or *f* (forbidden)
	Feature model reference *f1*

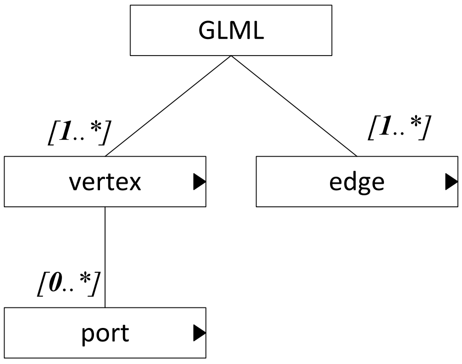


Fig. 2 GLML classifier with references to another feature model

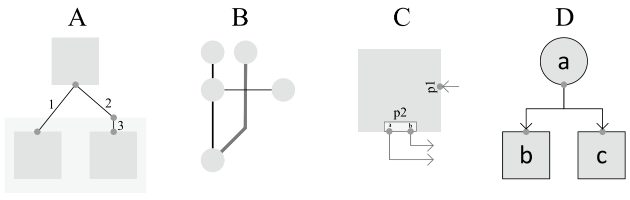


Fig. 3 Example GLML representations

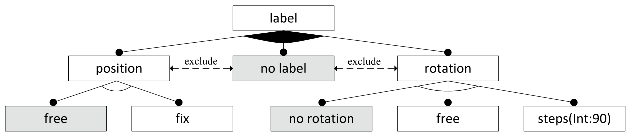


Fig. 4 Label classifier

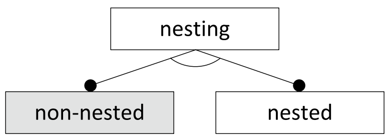


Fig. 5 Nesting classifier

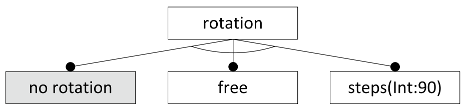


Fig. 6 Rotation classifier

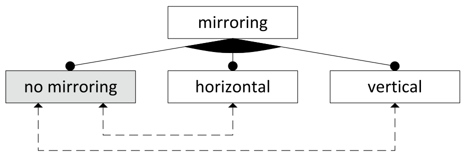


Fig. 7 Mirroring classifier

placement method. The complete classifier is displayed in appendix A.

Nesting Classifier

The nesting classifier defines the contents of vertices. Vertices can have content that is to be laid out (*nested*), have no content, or have non-laid out content (default: *non-nested*). Nested graphs pose complex requirements to layout algorithms, especially when vertices can be expanded and thus take up more space in a visual representation. Another difficulty can be the (edge) connection between inner and outer graph elements. The example A in Fig. 3 shows a nested graph with two different types of connection. Edge 1 crosses hierarchy boundaries of nodes. Edge 2 and edge 3, on the other hand, do not cross hierarchy boundaries (Fig. 5).

Rotation Classifier

The classifier characteristics for rotation and mirroring have importance for layout algorithms that place vertices. Rotation and mirroring can enable fewer crossings and bends of edges (Fig. 6).

If rotations are allowed at all, they can be allowed in *steps* of 90° or *free*.

Mirroring Classifier

If mirroring vertices is allowed at all (any classifier characteristic except the default *no mirroring*), the following features can be set: horizontal mirroring allowed, vertical mirroring allowed, horizontal and vertical mirroring allowed (Fig. 7).

Placement Classifier

The placement classifier is a simple string without subfeatures. It is not subset further because there are many different placement methods extended by further specializations and parameterizations. For example, [72] shows 200 different ways to visualize trees.[2]

The classifier is designed as a string to give a short impression of what types of placement are preferred for the GLML. This includes general placements, like *tree*, *layer*, *series–parallel*, *organic*, *grid*, *circular,* or domain-specific placements, like *UML-class*, *wiring diagram*, etc. The selection of a placement method is also greatly influenced by the other classifiers of a GLML.

Port Classification

The occurrence of ports in GLML as well as their properties have an enormous influence on layout algorithms. Especially restrictions regarding the position of ports influence the routing of edges. The complete classifier is displayed in appendix B.

Position Classifier

The port position is a defining characteristic of GLML. Many technical languages prescribe a fixed position on the vertices because the position has a semantic component. Changes in port positioning can lead to confusing diagrams or incorrect interpretations.

On the other hand, restricted port positions constrain the degrees of freedom for layout algorithms. Aesthetic criteria, like the minimization of crossings and bends, will be de-emphasized. The diagram layout loses readability. The default position of ports is *free*. If the feature is not explicitly denoted for a GLML, the ports in question can be placed freely on the vertex.[3]

The feature *side* can assign port positions to one or multiple sides of the vertex. The specific side is noted by a reference and becomes relevant when vertices are mirrored or rotated. *Local* defines the sides in relation to the host vertex, and *global* defines them in relation to the diagram (Fig. 8).

[2] Not all of these tree visualizations are suitable for GLML, but the number of usable visualizations is still quite high (vertical trees, horizontal trees, multidirectional trees, radial trees, hyperbolic trees).

[3] Ports are usually placed on the vertex shape. Position *free* thus also means a free placement on the vertex shape.

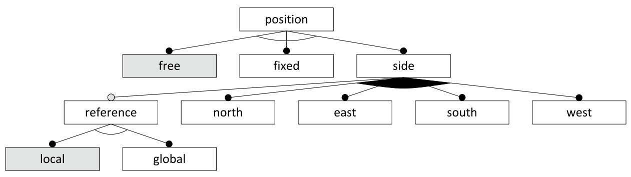


Fig. 8 Port position classifier

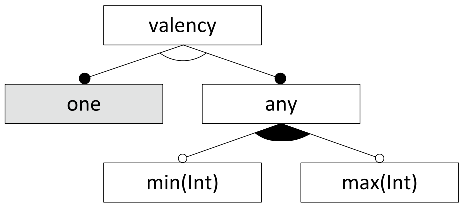


Fig. 9 Port valency classifier

The UML diagram in Fig. 1 shows an example of assigning ports to vertex sides, which allows the routing algorithm to route the aggregation edges without crossings.

Valency Classifier

The valency classifier describes how many edges can be connected to a single port. Its default value is *one*. Otherwise, the feature can be set to *any* and allow multiple connections. The number of connections can optionally be restricted by a minimum and/or maximum value. This also allows for a specific number with *min* = *max* (Fig. 9).

Restrictions on the number of connecting edges are typical for diagrams of technical systems. Circuit diagrams, for example, visualize terminals of electrical devices as ports and wires as edges. Terminals can be built to allow either one or two wires.[4]

The examples in Fig. 3 do not show their respective valency classifiers clearly. Example A can not have a valency of *one* for its upper port.

Nested Classifier

Some GLML have nested ports. These are described by the nested classifier. The superordinated port is referenced by a string. Example GLML C in Fig. 3 shows two nested ports a and b, with their classifiers being *a: nested(p2)* and *b: nested(p2)*.

[4] The classification scheme does not offer an explicit feature for vertex valency. Simple graphs in the classical model have their valency also defined using ports. Each vertex needs to have a port classifier without a symbol that is positioned on the vertex.

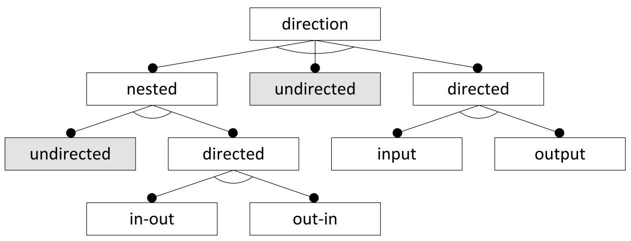


Fig. 10 Port direction classifier

SysML 1.6. uses nested ports for example in Block Definition Diagrams [8].

Direction Classifier

The direction classifier is used for directed graphs. It defines which end of a directed edge can be connected to a port. The default value is *undirected*, meaning that the port is directionless. With the feature set to *directed*, the port can connect to directed edges, and its direction is specified as *input* or *output*. The third type of port can occur in nested graphs. These ports represent the transit from an inner to an outer graph and can also be directed or undirected. In–out direction sets ports as input ports for the edges in the outer graph and output ports for the edges in the inner graph. Out–in is defined vice-versa (Fig. 10).

The first two examples in Fig. 3 have the following port direction classifiers.

GLML A: ports at the top vertex, left vertex, and bottom-right vertex: undirected (default); port direction between edges 2 and 3: nested.

GLML D: port at vertex a: directed- > output; ports at vertices b and c: directed- > input.

Edge Classification

The main classification features for edges are the number of possible edge connections, specifications for routing of the edges (e.g., straight or orthogonal), and the distinction between directed and undirected as well as hierarchic and non-hierarchic edges. The complete classifier is displayed in appendix C.

Structure Classifier

The structural composition of an edge is differentiated into 3 types: the default classifier is a port-to-port connection (*two ports*). The second type are edges that are connected to only *one port*. And the third type applies to GLML with hyperedges (*hyper*) (Fig. 11).

Some editors for graphical languages do not implement true hyperedges in their meta-model. Here hyperedges are

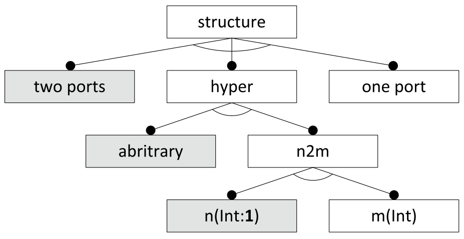


Fig. 11 Edge structure classifier

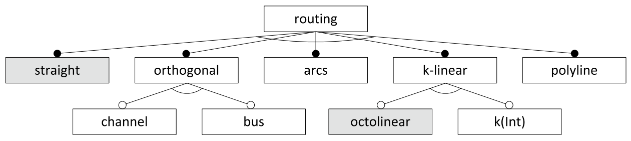


Fig. 12 Edge routing classifier

only visually represented by port-to-port connections routed onto buses or channels. This leads to the loss of structural information, especially if there are additional connectivity restrictions on ports (*n2m*).

A language in which edges can only be associated with one port is, e.g., IDEF0 [42, 43].[5]

The examples in Fig. 3 have the following edge structure.

A and B: two ports (default),

C: one port,

D: hyper.

Routing Classifier

The routing classifier defines the routing strategy. Every routing algorithm is controlled by its specific set of parameters, e.g., visualization of bend points (points, arcs, nothing), or handling of crossings. These parameters are not part of the routing strategy classifier.

The default value for the routing classifier is a straight routing without bends (*straight*). A common routing type is *orthogonal* routing. It was identified as a preferred characteristic for aesthetic graph drawing [62]. If the routing classifier is set to *orthogonal*, routing can only be achieved by horizontal or vertical line segments (Fig. 12).

Orthogonal routing can be further specialized by setting the features *channel* or *bus*. These features allow the GLML to emphasize characteristics of real world networks. Buses can be used to visualize the real properties of technical networks, e.g., power grids, and thus to better understand

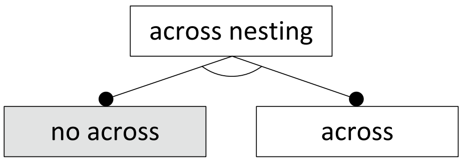


Fig. 13 Edge hierarchy nesting classifier

them [63]. The channel paradigm is used to visualize wires in circuit diagrams. Even for GLML not representing real-world networks, the visualization of hyperedges as buses can increase the clarity of the drawing. An example of this are hyperedges in UML diagrams [16].

Other routing classifiers are *arcs*, *k-linear,* and *polyline*. The latter two are mainly used in the map layout. So-called k-linear maps (cf. [73]) occur mainly in transportation maps, and the octolinear routing is the de-facto standard for such diagrams (cf. [27]). It is a routing of line segments whose angles to each other have an integer multiple of *360°/2k*, for example, $k = 2$ (orthogonal), $k = 3$ (hexalinear), and $k = 4$ (octolinear). Arc routing is used in classical graph drawing algorithms.

The examples in Fig. 3 have the following routing classifier.

A: straight (default),

B: k-linear- > octolinear (default),

C: orthogonal- > bus,

D: orthogonal- > channel.

Type Classifier

The type classifier implements the distinction of edge types and enables the classification scheme to classify *typed graphs.* Edge types can be represented by a label or by a specific graphic appearance in the concrete syntax of a GLML.

Type graphs and typed edges are important for routing algorithms. For example, crossings are evaluated differently if they occur between edges of the same or different types which can be used for the optimization of routings.

Example B in Fig. 3 shows a crossing between edges with different edge types. The edge types also have varied graphical representations.

Hierarchy Nesting Classifier

Edges in nested graphs can be routed across nesting hierarchy layers or only in their own layer (default). An example for the first case are SysML internal block diagrams, which link blocks across hierarchy layers with connectors (Fig. 13).

Example A in Fig. 3 shows an edge across the nesting hierarchy (edge 1) and two edges in their own layer (edges 2 and 3).

[5] Using the meta-model from Fig. 1 to visually represent edges connected only by a port, a symbolless pseudo vertex with a symbolless port must be created.

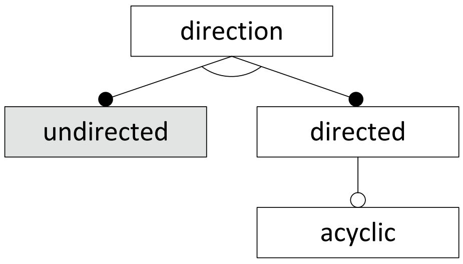


Fig. 14 Edge direction classifier

Direction Classifier

The direction classifier implements direction information for edges, with undirected as default. Directed edges have the optional feature *acyclic*. This feature can be important for layout algorithms, especially tree layouts or hierarchical layouts. Some common algorithms for these types of graphs can not handle cycles or need a special preprocessing step to eleminate cycles (Fig. 14).

The examples in Fig. 3 have the following direction classifier.

A and B: unirected (default),

C and D: directed.

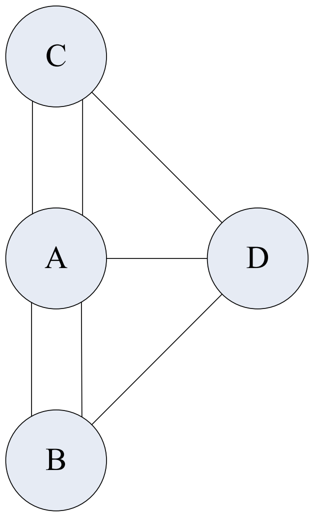


Fig. 15 Layout of the Königsberg bridge problem

```
GLML:Königsberg
   vertex:district
      placement = „orthogonal‟
   edge:bridge
```

Fig. 16 Classifier of the Königsberg graph example

Examples

In this section, the classification scheme presented for the concrete syntax is applied to three examples: a simple graph model, the UML diagram from Fig. 1, and a DSL. For the second example, the description of the classification is more detailed.

Simple Graph Model

Figure 15 shows a simple graph of the famous Königsberg bridge problem [74]: the vertices A, B, C, and D represent the districts, and the edges represent the seven bridges of Königsberg, drawn as a graph according to the classification.

For the simple graph model consisting of vertices, (undirected) edges, and no labels, combined with the additional request for orthogonal placement, the classification is in Fig. 16.

UML Class Diagram

The UML is very vague in defining the concrete syntax for diagrams. Tool developers are tasked with defining a concrete syntax for diagrams and implementing it in software tools via layout algorithms.

The UML class diagram in Fig. 1 is used as an example with the following concrete syntax:

1. Classes should be laid out in a „UML style“: Classes should be arranged in such a way that the inheritance hierarchy is taken into account. Classes with common aggregation and association relations should be placed close together.
2. Hyperedges are to be used for the inheritance relationships.
3. All edges must be routed orthogonally.
4. The ports for the inheritance relationship must be arranged in such a way that the ports on the base classes are placed on the lower outer border and the ports for the derived classes are placed on the upper outer border of vertices.
5. All other ports can be placed freely.
6. Ports of aggregation and association relations have labels (describing the multiplicities).
7. All ports are supposed to be evenly distributed on the outer contour of a vertex.

This results in the following characteristics of vertices, ports, and edges for the UML diagram in Fig. 1.

Vertices: The example UML-class-diagram has only one type of vertex (class), and the placement feature is *UML class*.

Ports can be separated into 5 types:

1. Ports of the base classes to which the edges for the inheritance relation are connected.
2. Ports of the derived classes to which the edges for the inheritance relation are connected.
3. Ports of the owning classes to which edges for the aggregation relation are connected.
4. Ports of the not owning classes to which edges for the aggregation relation are connected.
5. Ports for association relations.

Edges can be separated into 3 types:

1. Edges for inheritance relations are orthogonal hyperedges.
2. Edges for aggregation relations are port-to-port connections to be routed orthogonally.
3. Edges for association relations have the same classifier as aggregation relations.

This results in the classification of the concrete syntax in Fig. 17 for the UML diagram in Fig. 1.

This classification for a concrete syntax for UML class diagrams can be used to develop suitable layout algorithms or to reuse or adapt existing algorithms. For example, a static automatic layout for this graphical language is provided in [75], whereas [38] provide dynamic layout support for orthogonally routed hyperedges with layout stability.

Parameter Map of CAD-Model

The third example is the concrete syntax of a DSL. The model in Fig. 18 represents the parametric relationships of a 3D-CAD model of a deep drawing tool. The vertices represent the elements of the CAD model and their hierarchical structure. Identical model components are represented by a thick connection (e.g., the green connection between the occurrences of "punch assembly" in Fig. 18). The central vertex of the parameter map represents a specific parametric relationship (here a formula) between input variables (top) and output variables (bottom) [76].

The concrete syntax of the DSL can be described by the classifier in Fig. 19.

All vertices have the default properties of the classifier, but three types of ports: one type for the identity relationship with fixed positions and two types for the input ports and for the output ports (index 3) to connect directed edges between parameter vertices and formula.

```
GLML:UML
   vertex:class
      placement = „UML class‟
      port:inheritance-out
         position
            side
               south = m
         direction
            directed
               output = m
      port:inheritance-in
         position
            side
               north = m
         direction
            directed
               input = m
      port:aggregation-out
         position
            side
               south = m
               east = m
               west = m
         direction
            directed
               output = m
         label
            position
               fix = m
      port:aggregation-in
         position
            side
               north = m
               east = m
               west = m
         direction
            directed
               output = m
         label
            position
               fix = m
      port:association
         label
            position
               fix = m
   edge:inheritance
      structure
         hyper = m
      routing
         othogonal = m
            bus = m
      direction
         directed = m
      type = „derivation‟
   edge:aggregation
      routing
         othogonal = m
      direction
         directed = m
      type = „aggregation‟
   edge:association
      routing
         othogonal = m
      type = „association‟
```

Fig. 17 Classifier of the UML class diagram in Fig. 1

In accordance with the definition of ports, there are two types of edges. The edges that map the identity relationship have the default features. The edges between parameter vertices and formula vertices are directed hyperedges that are routed orthogonally.

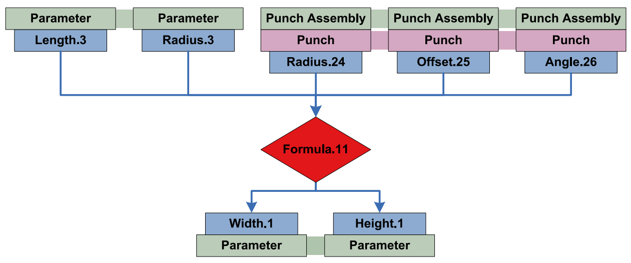


Fig. 18 Parameter map of a parametric 3D-CAD model for deep drawing tools [76]

```
GLML:Parameter Map
   vertex:any
      placement = „tree“
      port:parametric relation-out
         position
            fixed = m
         direction
            directed
               output = m
      port:parametric relation-in
         position
            fixed = m
         direction
            directed
               input = m
      port:identity relation
         position
            side
               east = m
               west = m
   edge:parametric relation
      structure
         hyper = m
      routing
         othogonal = m
            bus = m
      direction
         directed = m
   edge:identity relation
```

Fig. 19 Classifier of the parameter map in Fig. 18

Layout Algorithm Reuse

The main idea behind the design and development of the presented classification scheme was to support layout algorithm reuse. With a classifier describing a new GLML, it should be possible to find a matching layout algorithm (or multiple layout algorithms) that fulfills (or fulfill) the requested feature set.

Classifying layout algorithms as opposed to GLML is a different task because implemented layout methods typically have multiple options and parameters influencing their behavior. That means a single layout algorithm covers a wide array of different GLML classifiers. Feature diagrams can still be adapted to develop a classifier for layout algorithms, but this development is not in the scope of this paper.

Matching a GLML to layout algorithms can be done manually. Defining the classifier for a given GLML provides a good understanding of the relevant layout features. By examining the available layout algorithms, it is possible to choose the best solution for the GLML. In the best case, an existing layout algorithm can be used out-of-the-box or with only small adaptations.

In [11] three specific classifiers (*structure* and *routing* for edges and *position* for ports) were transferred to corresponding classifiers for layout algorithms. Furthermore, a mapping operator, which allows mapping between GLML and layout methods, was established. The respective classifiers for a GLML from the domain of process modeling of body-in-white production and for a fictitious layout method presented in [11] as well as a layout algorithm from [9] have been constructed. Applying the mapping operator from [11] then shows the fit between language and algorithms.

A worthwhile advancement is an automatic matching process. The classifiers for GLML and layout algorithms can be made machine-readable, e.g., in XML or JSON formats. Then it would be possible to enable existing software frameworks, like Eclipse GMF, to suggest layout algorithms for new GLML. Another approach would be the development of a “marketplace” for GLML layout, that facilitates the matching by storing classifier information.

Using classifiers to link GLML with layout methods has several benefits. The first is a faster and more efficient development of software tools implementing on GLML. Developers and researchers can concentrate their efforts on the GLML itself instead of building graph drawing tools. The second is obtaining better software tools in general because good interaction support and beautiful diagrams rely on good layout algorithms. Another benefit is the better propagation of research results. Only layout algorithms that are easily searchable and findable can be reused for different domains or contexts.

Conclusion and Further Work

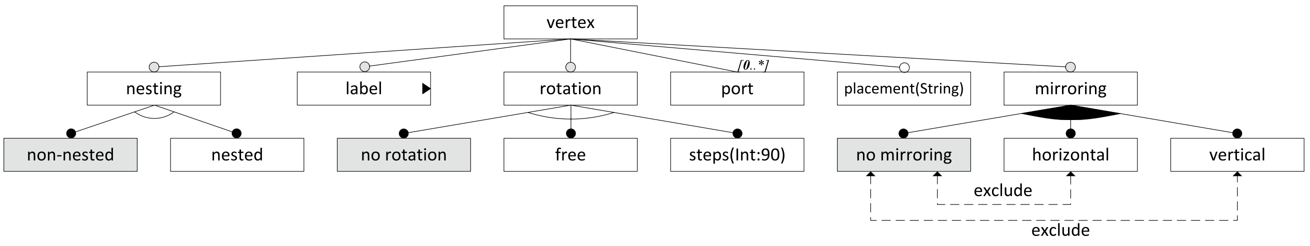
The usability of a GLML depends on good layout. Even users with limited modeling experience can build, modify, and understand complex models when supported by sensible layout algorithms. Because GLML appear in the form of GPML and DSL, they are important artifacts in both MBD tools and technical engineering tools. GLML have adapted many layout methods for graphs, especially for static drawings. Still, developing, reusing, and adapting graph drawing algorithms is very costly for tool developers due to the large variety of structurally different GLML. Dynamic layout algorithms are of greater importance than static layout algorithms in modeling tools, while their development is underrepresented in tool development.

The developed classification scheme for the concrete syntax offers the possibility to better relate GLML and layout methods. It shows a way to the reuse of layout algorithms and intra-language compatibility of GLML. Tool development for MDD is made more efficient by applying the presented approach. Other possible next steps would be to classify existing languages (e.g., GPML) and layout methods to facilitate mutual mapping.

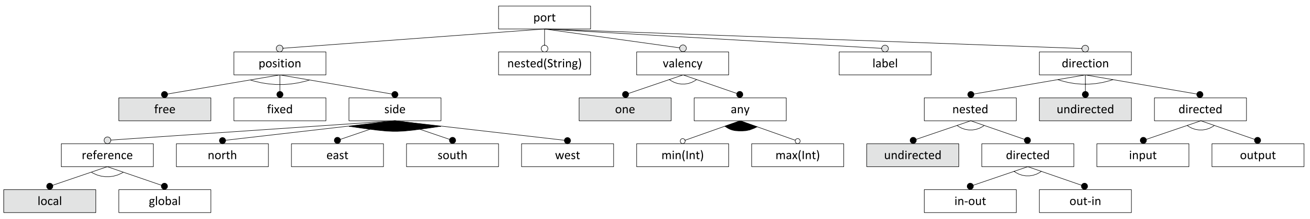
The classification scheme can be extended in further work. An important aspect is the design of a classifier set for layout algorithms [11]. This other side of the coin is needed to enable an automatic or assisted matching between the language design and layout implementation.

Furthermore, model-to-model transformations between different GLML with distinct classifiers are to be investigated. The classification scheme highlights relevant differences between GLML and supports the design of transformations in this way. With the help of these model-to-model transformations, even more layout methods, namely those for another class of GLML, can be reused.

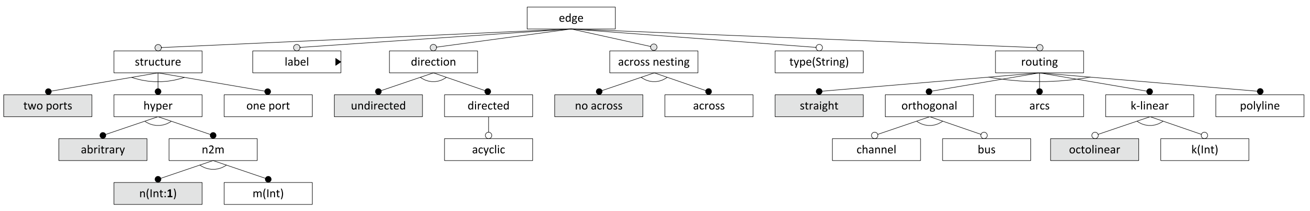
Appendix 1: Vertex Classifier



Appendix 2: Port Classifier



Appendix 3: Edge Classifier



Acknowledgements The authors thank the German Research Foundation (DFG) for the partly financial support of the research project “Method for the Model-Driven Design of Deep Drawing Tools” (project number BA 6300/1-3). This R&D project of the Society for the Advancement of Applied Computer Science (GFaI) is partly supported by the funding program Innovation Competence (INNO-KOM) of the German Federal Ministry for Economic Affairs and Climate Action (BMWK), based on a resolution of the German Parliament (project number 49VF200014).

Funding This study was partly funded by German Research Foundation (DFG) (Grant number BA 6300/1-3) and partly funded by German Federal Ministry for Economic Affairs and Climate Action (BMWK) (Grant number 49VF200014).

Data availability All data generated or analysed during this study are included in this published article.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with animals performed by any of the authors.

References

1. Tesla N. Apparatus for the utilization of radiant energy. Specification. https://patents.google.com/patent/US685957A/en (1901).
2. Karsai G, Krahn H, Pinkernell C, Rumpe B, Schindler M, Völkel S. Design guidelines for domain specific languages. In: Proceedings of the 9th OOPSLA workshop on domain-specific modeling (DSM' 09), vol. (2009).
3. van der Linden D, Hadar I, Zamansky A. What practitioners really want: requirements for visual notations in conceptual modeling. Softw Syst Model. 2019;18:1813–31. https://doi.org/10.1007/s10270-018-0667-4.
4. Gupta R, Jansen N, Regnat N, Rumpe B. Design guidelines for improving user experience in industrial domain-specific modelling languages (2022).
5. Binucci C, Brandes U, Dwyer T, Gronemann M, von Hanxleden R, van Kreveld M, Mutzel P, Schaefer M, Schreiber F, Speckmann B. 10 reasons to get interested in graph drawing. In: Steffen B, Woeginger G, editors. Computing and software science. Lecture notes in computer science, vol 10000. Cham: Springer; 2019. p. 85–104. https://doi.org/10.1007/978-3-319-91908-9_6.
6. Damasceno CDN, Strüber D. Quality guidelines for research artifacts in model-driven engineering (2021).
7. OMG UML 2.5.1: Unified Modeling Language, v2.5.1 (2017). https://www.omg.org/spec/UML/2.5.1/PDF.
8. OMG SysML 1.6: Systems Modeling Language v1.6 (2019). https://www.omg.org/spec/SysML/1.6/.
9. Eclipse Foundation: Eclipse Layout Kernel. Graph Data Structure (2021). https://www.eclipse.org/elk/.
10. Wrobel G, Scheffler R. Classification scheme for the concrete syntax of graph-like modeling languages for layout algorithm reuse. Setúbal: SCITEPRESSScience and Technology Publications; 2022. https://doi.org/10.5220/0010913400003119.
11. Wrobel G, Scheffler R. Classification and mapping of layout algorithms for usage in graph-like modeling languages 4th international workshop on modeling language engineering. https://doi.org/10.1145/3550356.3561559.
12. Schuette R, Rotthowe T. The guidelines of modelling—an approach to enhance the quality in information models. In: Ling T-W, editor. Conceptual modelling—ER ’98 17th international conference on conceptual modeling, Singapore, November 16–19, 1998, Proceedings. Lecture notes in computer science ser, vol 1507. Berlin: Springer; 1998. p. 240–54. https://doi.org/10.1007/978-3-540-49524-6_20.
13. Tamassia R, Di Battista G, Batini C. Automatic graph drawing and readability of diagrams. IEEE Trans Syst Man Cybern. 1988;18:61–79. https://doi.org/10.1109/21.87055.
14. Purchase HC, Cohen RF, James M. Validating graph drawing aesthetics, vol. 1027. Berlin: Springer; 1995. p. 435–46. https://doi.org/10.1007/BFb0021827.
15. Purchase H. Which aesthetic has the greatest effect on human understanding? In: Di Battista G, editor. Graph drawing. Lecture notes in computer science. Berlin: Springer; 1997.
16. Purchase HC, Allder J-A, Carrington D. User preference of graph layout aesthetics: a UML study. In: Goos G, Hartmanis J, van Leeuwen J, Marks J, editors. Graph drawing. Lecture notes in computer science, vol. 198. Berlin: Springer; 2001. p. 5–18. https://doi.org/10.1007/3-540-44541-2_2.
17. Ware C, Purchase H, Colpoys L, McGill M. Cognitive measurements of graph aesthetics. Inf Vis. 2002;1:103–10. https://doi.org/10.1057/palgrave.ivs.9500013.
18. Eichelberger H. Aesthetics of class diagrams. In: Proceedings. First international workshop on visualizing software for understanding and analysis, 26 June, 2002, Paris, France, p. 23–31. IEEE Computer Society, Los Alamitos, Calif (2002). https://doi.org/10.1109/VISSOF.2002.1019791.
19. Bergström G, Hujainah F, Ho-Quang T, Jolak R, Rukmono SA, Nurwidyantoro A, Chaudron MRV. Evaluating the layout quality of UML class diagrams using machine learning. J Syst Softw. 2022;192:111413. https://doi.org/10.1016/j.jss.2022.111413.
20. Lübke D, Ahrens M, Schneider K. Influence of diagram layout and scrolling on understandability of BPMN processes: an eye tracking experiment with BPMN diagrams. Inf Technol Manage. 2021;22:99–131. https://doi.org/10.1007/s10799-021-00327-7.
21. Dikici A, Turetken O, Demirors O. Factors influencing the understandability of process models: A systematic literature review. Inf Softw Technol. 2018;93:112–29. https://doi.org/10.1016/j.infsof.2017.09.001.
22. Di Battista G, Eades P, Tamassia R, Tollis IG. Algorithms for drawing graphs: an annotated bibliography. Comput Geom. 1994;4:235–82. https://doi.org/10.1016/0925-7721(94)00014-x.
23. Di Battista G, Eades P, Tamassia R, Tollis IG. Graph drawing. Algorithms for the visualization of graphs. Upper Saddle River: Prentice Hall; 1999.
24. Schulze CD, Spönemann M, von Hanxleden R. Drawing layered graphs with port constraints. J Vis Lang Comput. 2014;25:89–106. https://doi.org/10.1016/j.jvlc.2013.11.005.

25. Poon S-H, Shin C-S, Strijk T, Uno T, Wolff A. Labeling points with weights. Algorithmica. 2004;38:341–62. https://doi.org/10.1007/s00453-003-1063-0.
26. Wolff A. The map-labeling bibliography (2009). i11. www.iti.kit.edu/~awolff/map-labeling/bibliography/.
27. Wu H-Y, Niedermann B, Takahashi S, Roberts MJ, Nöllenburg M. A survey on transit map layout - from design, machine, and human perspectives. Comput Graph Forum J Eur Assoc Comput Graph. 2020;39:619–46. https://doi.org/10.1111/cgf.14030.
28. Pleßow M, Simeonov PL. Netlike schematics and their structure description. In: Workshop on informatics in industrial automation, p. 144–163 (1989).
29. May M, Kluge S, Pleßow M, Sieck J, Vigerske W. A review on block diagram layout. In: Proceedings of 5th IFAC symposium on computer-aided design in control systems (CADCS'91) (1991).
30. May M. Computer-generated multi-row schematics. Comput Aided Des. 1985;17:25–9. https://doi.org/10.1016/0010-4485(85)90007-7.
31. May M, Mennecke P. Layout of schematic drawings Systems Analysis Modelling Simulation.
32. Iwainsky A, Kaiser D, May M. Computer graphics and layout design in documentation processes. Comput Graph. 1990;14:377–88. https://doi.org/10.1016/0097-8493(90)90058-6.
33. Wrobel G, Ebert R-E, Pleßow M. Graph-based engineering systems—a family of software applications and their underlying framework. Electronic communications of the EASST, Volume 6: graph transformation and visual modeling techniques 2007, vol 6 (2007). https://doi.org/10.14279/tuj.eceasst.6.50.
34. Augenstein E, Wrobel G, Kuperjans I, Plessow M. TOP-energy-computational support for energy system engineering processes. In: Proceedings of the 1rst IC-SCCE. Athen (2004).
35. Marchenko M, Behrens B-A, Wrobel G, Scheffler R, Pleßow M. A New Method of visualization and documentation of parametric information of 3D CAD models. Comput Aided Des Appl. 2011;8:435–48. https://doi.org/10.3722/cadaps.2011.435-448.
36. Scheffler R, Murugan VP, Wrobel G, Pleßow M, Koch S, Buse C, Behrens B-A. Graphical modelling of a meta-model of CAD models for deep drawing tools. INCOSE Int Symp. 2016;26:1090–104. https://doi.org/10.1002/j.2334-5837.2016.00213.x.
37. Zeitler J, Goetze B, Fischer C, Franke J (ed) Integration of semi-automated routing algorithms for spatial circuit carriers into computer-aided design tools. Fürth (2014).
38. Helmke S, Goetze B, Scheffler R, Wrobel G. Interactive, orthogonal hyperedge routing in schematic diagrams assisted by layout automatisms. In: Basu A, Stapleton G, Linker S, Legg C, Manalu E, Viana P (eds) Diagrammatic representation and inference. 12th international conference, diagrams 2021. Virtual, September 28–30, 2021, proceedings. Lecture notes in computer science book series (LNCS), Part of the lecture notes in computer science book series (LNCS, volume 12909); also part of the lecture notes in artificial intelligence book sub series (LNAI, volume 12909), p. 20–27. Springer Nature, Cham (2021). https://doi.org/10.1007/978-3-030-86062-2_2.
39. Barzdins J, Kalnins A. Metamodel specialization for graphical language and editor definition. BJMC. 2016;4:910–33. https://doi.org/10.22364/bjmc.2016.4.4.20.
40. yWorks GmbH: The Graph Model (2022). https://docs.yworks.com/yfileshtml/#/dguide/graph.
41. USAS Y14.15-1966: Electrical and electronics diagrams. The American Society of Mechanical Engineers, New York (1966).
42. IDEF N. Integrated DEFinition Methods (IDEF). IDEF Family of Methods (2021). https://www.idef.com/.
43. Menzel C, Mayer RJ. The IDEF Family of languages. In: Bernus P, Mertins K, Schmidt G (eds) Handbook on architectures of information systems. Springer eBook collection computer science, pp 215–249. Springer, Berlin (2006). https://doi.org/10.1007/3-540-26661-5_10
44. IEC 61131-3:2013: International electrotechnical commission IEC 61131-3:2013. Programmable controllers—Part 3: Programming languages (2014).
45. Bork D, Karagiannis D, Pittl B. A survey of modeling language specification techniques. Inf Syst. 2020;87:101425. https://doi.org/10.1016/j.is.2019.101425.
46. Kühne T. Matters of (meta-) modeling. Softw Syst Model. 2006;5:369–85. https://doi.org/10.1007/s10270-006-0017-9.
47. Vaupel S, Taentzer G, Harries JP, Stroh R, Gerlach R, Guckert M. Model-driven development of mobile applications allowing role-driven variants. In: Dingel J, Schulte W, Ramos I, Abrahão S, Insfran E, editors. Model-Driven engineering languages and systems. Lecture notes in computer science, vol 8797. Cham: Springer International Publishing; 2014. p. 1–17. https://doi.org/10.1007/978-3-319-11653-2_1.
48. Chu M-H, Dang D-H, Nguyen N-B, Le M-D, Nguyen T-H. USL sfvmkf vkmvkb. In: Thang HQ, Hu Z, Bui M, Sikdar B, Ide I, Binh HTT, Engchuan W, Sang DV, Oanh NT, editors. Proceedings of the Eighth International Symposium on Information and Communication Technology. New York: Springer; 2017. p. 401–8. https://doi.org/10.1145/3155133.3155194.
49. Maillard S, Smeda A, Oussalah M. COSA: An architectural description meta-model. In: Filipe J (ed) Programming languages, distributed and parallel systems, knowledge engineering, special session on metamodeling—utilization in software engineering (MUSE). Proceedings of the Second International Conference on Software and Data Technologies Barcelona, Spain, July 22–25, 2007, p. 445–448. INSTICC, Setúbal (2007). https://doi.org/10.5220/0001338404450448.
50. GMF: Graphical Modeling Framework|The Eclipse Foundation (2021). https://www.eclipse.org/modeling/gmp/.
51. OBEO: Obeo Designer (2021). https://www.obeodesigner.com/en/solutions.
52. Sirius: Sirius Overview (2021). https://www.eclipse.org/sirius/overview.html.
53. MetaEdit+: MetaEdit+ Domain-Specific Modeling (DSM) environment (2021). https://www.metacase.com/products.html.
54. Microsoft DSL: Modeling SDK for Visual Studio-Domain-Specific Languages (2021). https://docs.microsoft.com/en-us/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages?view=vs-2022.
55. Cooper J, de La Vega A, Paige RF, Kolovos D, Michael B, Brown C, Sanchez Pina BA, Hoyos Rodriguez H. Model-based development of engine control systems: experiences and lessons learnt. In: ACM/IEEE 24th international conference on model driven engineering languages and systems (2021).
56. Guo H, Huang J, Laidlaw DH. Representing uncertainty in graph edges: an evaluation of paired visual variables. IEEE Trans Visual Comput Graph. 2015;21:1173–86. https://doi.org/10.1109/TVCG.2015.2424872.
57. Meidiana A, Hong S-H, Eades P. New quality metrics for dynamic graph drawing. In: Auber D, Valtr P (eds) Graph drawing and network visualization. 28th international symposium, GD 2020, Vancouver, BC, Canada, September 16–18, 2020 : revised selected papers. Lecture notes in computer science SL3-Information Systems and Applications, incl. Internet/Web, and HCI, vol 12590, p. 450–465. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-68766-3_35.
58. Archambault D, Purchase HC. Mental map preservation helps user orientation in dynamic graphs. In: Didimo W, Patrignani M (eds) Graph drawing. 20th international symposium, GD 2012, Redmond, WA, USA, September 19–21, 2012, Revised Selected

Papers. SpringerLink Bücher, vol. 7704, pp. 475–486. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-36763-2_42.

59. Eades P, Lai W, Misue K, Sugiyama K. Preserving the mental map of a diagram. In: Proceedings of compugraphics '91, p. 24–33 (1991).
60. Kobourov SG, Mchedlidze T, Vonessen L. Gestalt principles in graph drawing. In: Di Giacomo E, Lubiw A, editors. Graph drawing and network visualization. Lecture notes in computer science, vol 9411. Cham: Springer; 2015. https://doi.org/10.1007/978-3-319-27261-0_50.
61. Taylor M, Rodgers P. Applying graphical design techniques to graph visualisation. In: Banissi E (ed) Proceedings/nineth international conference on information visualisation, 2005. 06–08 July 2005, [London, England, pp. 651–656. IEEE Computer Society, Los Alamitos, Calif. (2005). https://doi.org/10.1109/IV.2005.19.
62. Purchase HC. Metrics for graph drawing aesthetics. J Vis Lang Comput. 2002;13:501–16. https://doi.org/10.1006/jvlc.2002.0232.
63. Wrobel G, Scheffler R, Kehrer T. Rethinking the traditional design of meta-models: layout matters for the graphical modeling of technical systems 2021 ACM/IEEE 24th international conference on model driven engineering languages and systems companion (MODELS-C), p. 351–360 (2021). https://doi.org/10.1109/MODELS-C53483.2021.00058.
64. Saffrey P, Purchase HC. The 'Mental Map' versus 'Static Aesthetic' compromise in dynamic graphs : a user study. In: Plimmer B, Weber G (eds) User interfaces 2008, ninth Australasian user interface conference, AUIC 2008, Wollongong, NSW, Australia, January 2008. CRPIT, vol. 76, pp. 85–93. Australian Computer Society (2008).
65. Kang K, Cohen S, Hess J, Novak W, Peterson A. Feature-oriented domain analysis (FODA) feasibility study. Pittsburgh, PA (1990).
66. Czarnecki K, Eisenecker UW, Eisenecker U. Generative programming. Methods, tools, and applications. Boston: Addison Wesley; 2000.
67. Czarnecki K. Generative programming: principles and techniques of software engineering based on automated configuration and fragment-based component models. Germany: Ilmenau; 1999.
68. Riebisch M, Böllert K, Streitferdt D, Philippow I. Extending feature diagrams with Uml multiplicities conference on integrated design and process technology (IDPT 2002). Pasadena, California, USA (2002).
69. Czarnecki K, Kim CHP (eds): Cardinality-based feature modeling and constraints: a progress report (2005).
70. EN 60617-2: Graphical symbols for diagrams. Part 2: Symbol elements, qualifying symbols and other symbols havin general application, vol. 01.080.30; 29.020 (1997).
71. Imhof E. Die Anordnung der Namen in der Karte. Int Yearb Cartogr. 1962;20:93–129.
72. Lima M. Book of trees. Visualizing branches of knowledge. New York: Princeton Architectural Press; 2014.
73. Nickel S, Nöllenburg M. Towards data-driven multilinear metro maps (2019).
74. Euler L. Solutio problematis ad geometriam situs pertinentis. Commentarii academiae scientiarum Petropolitanae, vol. 128–140 (1735).
75. Eiglsperger, M.: Automatic layout of UML class diagrams. Tübingen (2003)
76. Scheffler R, Koch S, Wrobel G, Pleßow M, Buse C, Behrens B-A. Modelling CAD models. Method for the model driven design of CAD models for deep drawing tools 4th international conference on model-driven engineering and software development (MODELSWARD), pp 377–383 (2016). https://doi.org/10.5220/0005799403770383.