

# Classification and Mapping of Layout Algorithms for Usage in Graph-Like Modeling Languages

Gregor Wrobel

Graph Based Engineering Systems  
Society for the Advancement of Applied Computer  
Science  
Berlin, Germany  
wrobel@gfai.de  
0000-0003-4234-0794

Robert Scheffler

Graph Based Engineering Systems  
Society for the Advancement of Applied Computer  
Science  
Berlin, Germany  
scheffler@gfai.de  
0000-0002-3015-0099

## ABSTRACT

Graph-like modeling languages (GLML) are important modeling artifacts for software tools that are used in the environment of software and systems engineering, digital twins, and domain-specific modeling. Just as with textual languages, the concrete syntax is the representation of the language elements intended for humans and thus has a decisive influence on the comprehensibility and usability of the language.

The concrete syntax of GLML is often defined less precisely or in less detail when designing metamodels. While metamodels can be designed independently or even without concrete syntax, the latter is required for the actual usage of a graphical language. The layout and the interaction support for editing the language elements are commonly delegated to tool development. However, the development of modeling tools focuses on functionality such as model transformation and model execution, disregarding usability and handling. Low user acceptance then leads to niche applications and a limited number of users. The main reason for the lack of support for laying out and editing the language elements are complex integration challenges mainly concerning the development or adaptation of suitable layout procedures for GLML.

Some frameworks offer a remedy by providing layout procedures for GLML. However, GLML differ from each other concerning their concrete syntax. Even minor differences in the concrete syntax of two languages can make the desirability or only transferable through complex adaptations. Formal methods for matching the concrete syntax of a GLML with existing layout procedures as early as during the development of the language are missing.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

*MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada*

© 2022 Copyright is held by the owner/author(s).

ACM ISBN 978-1-4503-9467-3/22/10.

<https://doi.org/10.1145/3550356.3561559>

In this paper, we present a classification scheme for layout procedures for GLML. The classification scheme is based on a classification scheme we developed for the concrete syntax of GLML, and it contains significant features. We define a mapping procedure between the two classifiers. As a result, the mapping decides whether a layout procedure can be used for a GLML. Both the classification scheme and the process of mapping are demonstrated in a real-world example of a technical graphical domain-specific modeling language.

The presented approach supports the design of GLML and the tool development for GLML. It can be an important step towards automated tool development.

## CCS CONCEPTS

• Computing methodologies~Modeling and simulation~Model development and analysis~Modeling methodologies • Software and its engineering~Software notations and tools~Formal language definitions~Syntax • Human-centered computing~Visualization~Visualization techniques~Graph drawings

## KEYWORDS

Graph-Like Modeling Languages, Concrete Syntax of Modeling Languages, Layout-Algorithm Reuse, Designing and Implementing Modeling Languages

## 1 Introduction

Increasing digitization and the requirements to develop and master more and more complex systems have brought paradigms and approaches such as Model-Driven Engineering (MDE) [1] and Digital Twins [2] into the focus of research and industrial application. Essential artifacts for this are models. As a representation of systems throughout their life cycle, models are the most important interface between man and machine. Thus, models can facilitate the transition between informally sketched requirements or drafts to implementations, or they are the basis for simulations and optimizations. There are both textual and

graphical modeling languages, whereby the latter can have their particular strength in a clear representation of the different model elements and their interrelationships.

A considerable number of well-known graphical modeling languages that are widely used in industrial applications (e.g., automotive, avionics, or telecommunications) exist. These include standardized modeling languages such as UML [3], which can be tailored to specific domains using the UML profile mechanism, resulting in languages such as SysML [4] or MARTE [5]. In addition to UML-based languages, there are varieties of dedicated modeling languages with a narrower scope. Prominent examples of this are BPMN [6] and Matlab/Simulink [7]. These languages, also called General-Purpose Modeling Languages (GPML), have some general domain references (e.g., UML for modeling the structure and behavior of object-oriented software, Matlab/Simulink for simulation models), but they are also used for other specific domains. Their universality implies that their language constructs are not very specifically targeted to a particular domain. It is even the case that the concrete syntax of GPML is loosely defined [8]. This is also reflected in the users of these languages. Either they are (software) developers who are familiar with abstract language constructs through programming languages, or the users come from an academic environment and have familiarized themselves with a GPML for their academic work. The missing or weak definition of the concrete syntax even allows this user group freedom in the use of the language.

Another class of languages has a stronger concrete domain reference. Users of these domain-specific languages (DSL) are domain experts who often have less experience with abstract languages. In order to enable proper use of these languages, many of them have a more detailed description of their concrete syntax. There are some very old, widely used languages that are still in use, such as circuit diagrams for the representation of electrical circuits [9, 10]. Their concrete syntax, but also aesthetic criteria, as proclaimed in the field of graph drawing [11], and layout requirements are partly specified in great detail using standards.<sup>1</sup> On the one hand, this makes tool development easier; and on the other hand, it makes it easier for users to learn and master the language. Adherence to the standard means that once learned, it can be used repeatedly, and models can be exchanged. Such languages can justifiably be called successful because they have been used for decades.

In more recent language development, the human-centric aspects are less in focus. Model executability and model transformations are the core of MDE [12], and the abstract syntax definition of languages is the focus of language engineering.

According to our experience from the development of GLML for different application domains in engineering,<sup>2</sup> human factors

are one, if not the decisive, criterion for the successful establishment of a graphical modeling system (for technical systems) [8].<sup>3</sup> The central hypothesis stated in [8] is that the layout of GLML is the most important criterion for understanding, creating, and editing models. All three use cases are of great importance in the engineering process for technical systems. A modeling process starts with a blank sheet (creating). Through user interactions, the model grows; and, as it grows, it requires changes in the placement of vertices and the routing of connections<sup>4</sup> of a GLML. The models are not built all at once or by just one user. Thus, the model must also be understandable (understanding) and changeable (editing). Making each interaction efficient and effective to use requires good layout procedures that place vertices and route connections, ideally taking into account the users' mental map [16, 17].

A special field of graph theory deals with the placement of vertices and the routing of edges: graph drawing [18]. In the last decades, many algorithms for placing vertices (grid, trees, layered, series-parallel, organic, circular, etc.) and routing networks (straight lines, rectilinear lines, busses and channels, polylines, arcs, etc.) were developed there. In addition, special types of graphs were studied (port graphs, hypergraphs, nested graphs, labeled graphs), and algorithms for them were developed. So why not simply include the many layout methods of graph drawing in language engineering and integrate them into the tools?

From our point of view, there are two main reasons for this.

1. The classical graph model consisting of vertices and edges is not well suited as a metamodel for GLML, and
2. in language engineering, it is not clear which layout methods are suitable for the concrete syntax of a language. There are so many different types of graph models or graph-like metamodels and very many layout methods tailored for these models.

The first reason is mainly based on the fact that the vertices of GLML are not dimensionless vertices to which edges can be connected anywhere. In GLML, the vertices are two-dimensional objects, and their connection points often have semantics<sup>5</sup>. The concrete syntax of these connection points, e.g., whether they can be moved or exchanged, have an enormous impact on routing algorithms, for example, when it comes to minimizing intersections.

These aspects are already incorporated into the metamodels of some modeling frameworks. A number of models [19–22] distinguish between the core elements vertex, edge, and port.

This paper is dedicated to the second reason mentioned. We present an approach that can make statements about whether a layout procedure can be used structurally for a GLML as early as

<sup>1</sup> For example: "Lines between symbols should be horizontal or vertical with a minimum of line crossings, and with spacing to avoid crowding" [9].

<sup>2</sup> For example, flowchart diagrams for energy system engineering [13, 14] or parameter maps for parametric 3D-CAD models [15].

<sup>3</sup> In [8], graphical modeling languages for technical systems are explicitly the subject of consideration. The authors do not want to exclude a generalization to graphical modeling in general at this point, but they do not want to proclaim it either.

<sup>4</sup> The placement of vertices and routing of edges is what is called layout in this paper.

<sup>5</sup> For example, the connection points in UML activities are distinguished (symbolically) in that for the object flow a square symbolizes the connection point, but no symbol is used for the control flow. Another example are the connection points at symbols in circuit diagrams, which have a physical representation as connection points of cables and for which therefore a symbolic distinction is of immense importance.

during language engineering. The following questions are to be answered by this approach:

1. Can a layout method related to the concrete syntax of a graphical language be used for it?
2. Which layout algorithms are available for a GLML and which are not?
3. And which layout algorithms are no longer available because of a change in the concrete syntax of a GLML?<sup>6</sup>

To answer these questions, we present the following approach in this paper:

We define a classification scheme for the concrete syntax of GLML and a classification scheme for layout procedures. Both schemes are feature-based (Section 3), based on the metamodel core in Figure 1, and the features between the schemes correspond with each other (Section 4). The second step is to define a mapping between the two classifiers that can answer the questions described above. The approach is applied (Section 6) to the example described at the beginning (Section 2). The paper ends with a Section on related work (Section 7) and a summary and outlook (Section 8).

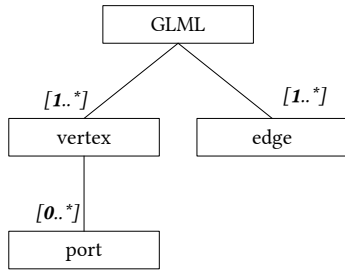


Figure 1: GLML Core Elements

## 2 Example Application

In this Section, we motivate the classification of the concrete syntax of GLML and the classification of layout procedures for GLML, and we describe the requirement to a mapping between the two classifiers using a real-world example.

The example in Figure 2 shows a process model for a body construction plant of an automotive manufacturer. The model describes how further components (*part 1.2* to *part 4.3*) are mounted to an initial component (*part 1.1*). The components are joined together in joining steps, which are represented by the larger vertices. A joining step contains a subgraph that describes the type and sequence of joining processes (e.g., welding, clinching) and that is to be displayed either expanded (*joining step 3.1* and *joining step 4.1*) or unexpanded (*joining step 2.1* and *joining step 5.1*) in the editor. The placement of the vertices in the process model reflects the semantics of the assembly. On the one hand, the arrangement of the components symbolizes the passage of the main component through the plant in the form of the horizontal arrangement of the vertices. On the other hand, the arrangement of the vertices at the top or bottom shows the mounting of their respective parts to the right and left of the main part.<sup>7</sup>

When editing a process model, a typical interaction is moving vertices in the graph. The act of moving vertices symbolizing components (part in Figure 2) can be divided into two cases. In case 1, the joining step that is to the right of vertex *part 1.1*, is not expanded. In case 2, it is expanded. In the following, we look at the two interactions and the expected layout result:

1. Moving part 1.3 above part 1.1:

The result should be a layout as in constellation *part 4.2*, *part 4.3*, and *joining step 5.1*. Thus, a suitable layout method must lay out the hyperedge connecting *part 1.1*, *part 2.1*, *part 1.3*, and *joining step 2.1* again.

2. Moving part 2.3 above joining step 2.1:

The result should be a layout as in the constellation *part 3.2*, *part 3.3*, and *joining step 4.1*. Thus, a suitable layout method must lay out the edge connecting *part 2.3* and *joining step 2.1* again. To get a clear routing (shortest possible paths, free of intersections, and few bends [11]), the port connecting *part 2.3* to *joining step 3.1* has to be moved and the edge between the vertices has to be re-routed.

A suitable interaction method must therefore be able to handle hyperedges and move pins. If both properties are given by the concrete syntax of the GLML, a layout method can be searched that supports these properties of the GLML.

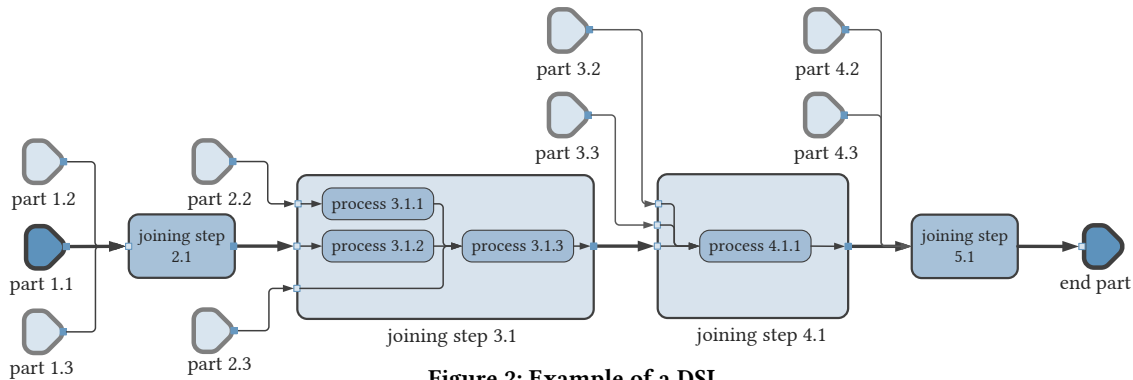


Figure 2: Example of a DSL

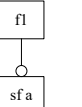
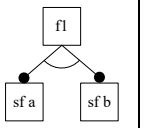
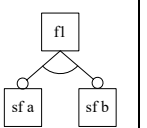
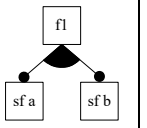
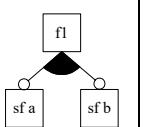
<sup>6</sup> This could have an influence on the definition of the concrete syntax of a language, if alternatives are possible, and more suitable layout algorithms are available for them.

<sup>7</sup> The arrangement follows domain-specific context. For more information see [8].

### 3 Notation

The classifying schemes in this paper use a notation based on feature diagrams. Feature diagrams were introduced as part of Feature-Oriented Domain Analysis (FODA) [23] and later expanded through various works. The notation used in this paper includes the extensions by [24].

**Table 1: Notation**

| # | Symbol  | Explanation  |
|---|---|--|
| 1 |    | Feature <i>f1</i> with optional solitary subfeature <i>sf a</i> . If the subfeature is set it can have the values <i>m</i> (mandatory) and <i>f</i> (forbidden).   |
| 2 |    | Feature <i>f1</i> with exactly one subfeature from a group. The specific subfeature that is set has the value <i>m</i> (mandatory). All other subfeatures have the value <i>f</i> (forbidden). This classifier type is not used for layout algorithms.   |
| 3 |   | Feature <i>f1</i> with exactly one subfeature from a group that is optional. The set subfeature has the value <i>m</i> (mandatory). All other subfeatures have the value <i>f</i> (forbidden). This classifier type is not used for layout algorithms.   |
| 4 |  | Feature <i>f1</i> with at least one subfeature of a group. The set subfeatures have the value:<br><ul style="list-style-type: none"> <li>– <i>m</i> (mandatory) for GLML,</li> <li>– <i>m</i> (mandatory) or <i>s</i> (supported) for layout algorithms.</li> <li>– All other subfeatures have the value:<br/> <ul style="list-style-type: none"> <li>– <i>f</i> (forbidden) for GLML,</li> <li>– <i>f</i> (forbidden) or <i>n</i> (not supported) for layout algorithms.</li> </ul> </li> </ul> |
| 5 |  | Feature <i>f1</i> with optionally one or more subfeatures of a group. The subfeatures have the value:<br><ul style="list-style-type: none"> <li>– <i>m</i> (mandatory) or <i>f</i> (forbidden) for GLML</li> <li>– <i>m</i> (mandatory), <i>f</i> (forbidden), <i>s</i> (supported), or <i>n</i> (not supported) for layout algorithms.</li> </ul>   |

The semantics described by the syntax of the feature diagrams states whether a feature must be specified for a language or layout method (lines 2 and 4) or can be specified optionally (lines 3 and 5). The values that the features can take (*mandatory* and *forbidden* for GLML, and additionally *supported* and *not supported* for layout algorithms) then refer to the instance of a feature in a concrete classifier.

### 4 Classification Scheme for the Concrete Syntax of GLML and Layout Algorithms

In [25], we introduced a classification scheme for the concrete syntax of GLML. The scheme distinguishes between vertices and edges as well as ports that are assigned to the vertices as essential components of a GLML. In subsequent work, we extended the classification scheme and transitioned it to the notation described above. In [25], only the concrete syntax of GLML is classified. Here, we now present a classification scheme for layout algorithms that has the same features as the classification scheme for the concrete syntax of GLML. This allows the development of a mapping between both classifiers.

The elements are classified by the following features, each of which is described by further subfeatures:

- Vertex: label, ports, nesting, rotation, mirroring, placement.
- Port: label, position, direction, nested, valency.
- Edge: label, structure, direction, across nesting, routing.

It should be noted that rendering aspects are not included in the classification scheme for layout algorithms. The concrete graphical appearance of diagram elements, e.g. colors or symbols, is not part of the scheme.

To show the idea of classifying concrete syntax of a GLML, classifying layout procedures, and performing the mapping between the two classifiers, we restrict ourselves to the part of the classifiers that are required for the example described at the beginning. Since the interaction of moving a vertex described there entails a routing procedure, it is reasonable to assume that the edge classifier is of particular importance for the example GLML shown in Figure 2.

Figure 3 shows the classifier for the feature structure and the classifier for the type of routing for edges as well as the classifier for the position of ports, each for the concrete syntax of a GLML (left side) and layout algorithm (right side). It is noteworthy that the top-level features have to be specified in the concrete syntax of GLML<sup>8</sup> and may be specified for the layout algorithms.

<sup>8</sup> In order to keep the actual classifiers compact, default values for individual features were specified in [25], so that not every feature has to be specified for GLML either

(default subfeatures are given, which adopt the value *mandatory*). For the sake of brevity, the default values have been omitted in this paper.

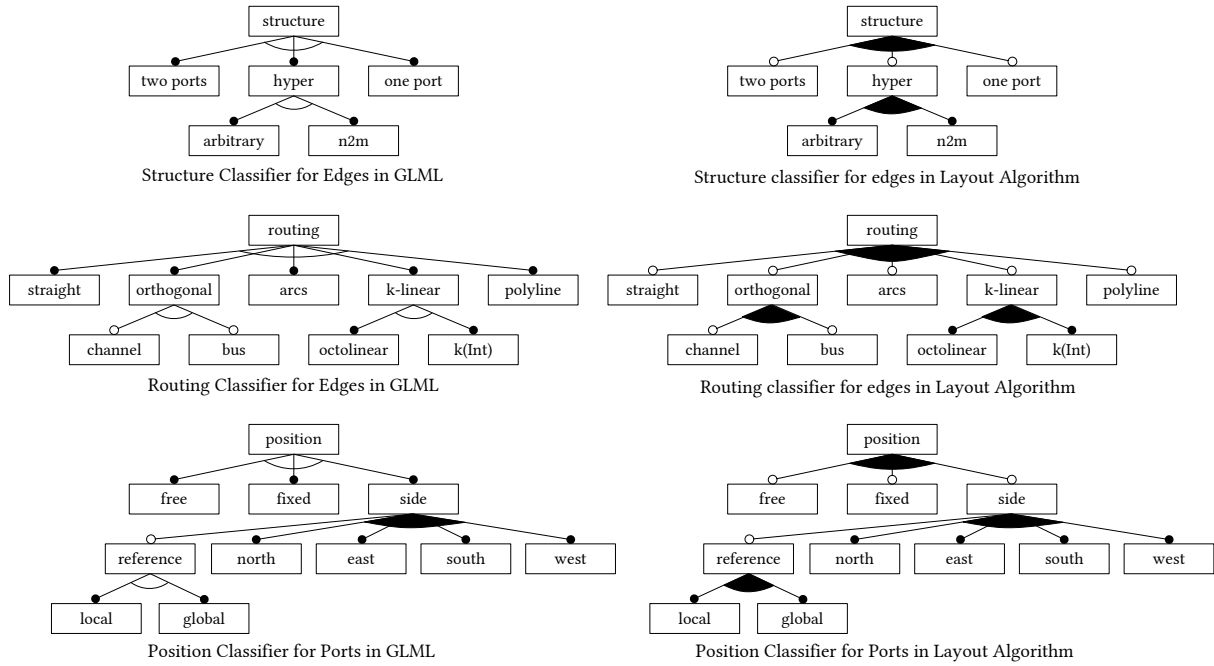


Figure 3: Excerpt of the Classifiers of Edges and Pins, for the Concrete Syntax of GLML and Layout Algorithms Respectively

## 6 Mapping of Concrete Syntax Classifier and Layout Algorithm Classifier

To facilitate the mapping between corresponding features  $f$  of a GLML and a layout algorithm, we define the logical operator  $\equiv$ . The operator can be applied by iterating through the features of the mapping candidates and selecting the appropriate row of Table 2.

**Definition** (Mapping): For a given classification scheme  $C$ , a mapping between a specific GLML  $G$  and a concrete layout algorithm  $L$  is defined as:

A GLML  $G$  and a layout algorithm  $L$  can be mapped,

or  $G \equiv L = \text{true}$ , if  $G.f \equiv L.f = \text{true}, \forall f \in C$ .

Conversely, a GLML  $G$  and a layout algorithm  $L$  cannot be mapped, or  $G \equiv L = \text{false}$ , if  $G.f \equiv L.f = \text{false}, \exists f \in C$ .

The semantics of the mapping definition are:

1. If a layout algorithm supports a feature, then the value of this feature in the GLML classifier is insignificant (case 1-3).
2. A feature that is *not supported* by a layout algorithm (case 4-6) cannot be set in the GLML classifier. The layout algorithm can still be used, if the GLML does not specify the feature at all (case 6).
3. If a feature is *mandatory* in a layout algorithm, it also has to be *mandatory* in the GLML classifier to form a valid mapping (case 7-8).
4. If a layout algorithm has a feature set as forbidden, then the same feature also has to be specified as forbidden in the

concrete syntax of the GLML (case 10-12).

5. If a feature is not specified for a layout algorithm, then it is irrelevant. The mapping operator is always true (case 13-15).

Table 2: Mapping Operator<sup>9</sup>

| #  | GLML.f                             | Layout.f   | GLML.f $\equiv$ Layout.f |
|----|------------------------------------|--|--------------------------|
| 1  | $m$                                | $s$  | <i>true</i>              |
| 2  | $f$                                | $s$  | <i>true</i>              |
| 3  | $\neg m \wedge \neg f = \emptyset$ | $s$  | <i>true</i>              |
| 4  | $m$                                | $n$  | <i>false</i>             |
| 5  | $f$                                | $n$  | <i>true</i>              |
| 6  | $\neg m \wedge \neg f = \emptyset$ | $n$  | <i>true</i>              |
| 7  | $m$                                | $m$  | <i>true</i>              |
| 8  | $f$                                | $m$  | <i>false</i>             |
| 9  | $\neg m \wedge \neg f = \emptyset$ | $m$  | <i>false</i>             |
| 10 | $m$                                | $f$  | <i>false</i>             |
| 11 | $f$                                | $f$  | <i>true</i>              |
| 12 | $\neg m \wedge \neg f = \emptyset$ | $f$  | <i>false</i>             |
| 13 | $m$                                | $\neg n \wedge \neg s \wedge \neg m \wedge \neg f = \emptyset$ | <i>true</i>              |
| 14 | $f$                                | $\neg n \wedge \neg s \wedge \neg m \wedge \neg f = \emptyset$ | <i>true</i>              |
| 15 | $\neg m \wedge \neg f = \emptyset$ | $\neg n \wedge \neg s \wedge \neg m \wedge \neg f = \emptyset$ | <i>true</i>              |

<sup>9</sup> For the feature value definition see Table 1:  $m$  (mandatory),  $f$  (forbidden),  $s$  (supported), and  $n$  (not supported)

**Remarks**  $\neg m \wedge \neg f$  in the GLML classifier means that this feature is not relevant to the language. It follows that any mapping is possible, regardless of the feature of the layout algorithm.

$\neg n \wedge \neg s \wedge \neg m \wedge \neg f$  in the layout algorithm classifier means that this feature is not relevant to the layout. Accordingly, any mapping is possible, regardless of the GLML feature.

In the following, we use  $\emptyset$  to shorten  $\neg n \wedge \neg s \wedge \neg m \wedge \neg f$  and  $\neg m \wedge \neg f$ .

$s$  or  $\emptyset$  in the layout classifier means that there is a possible mapping with any GLML classifier:  $(m \vee f \vee \emptyset) \equiv (s \vee \emptyset) = \text{true}$ .

We use *any* to shorten  $m \vee f \vee \emptyset$ .

Applying the operator to the combination of GLML and layout algorithm classifiers answers the question of a possible mapping.

## 6 Concrete Mapping Example

Here we show the relevant classifiers to accomplish a concrete mapping. The GLML is the example introduced in section 2, and the layout algorithm is one that supports the described layout interactions.

The classifier in Figure 5 is an excerpt of the full classifier for the concrete syntax of the modeling language in Figure 2. All the classifier features in Figure 3 as well as other, non-pictured, features (port->direction; edge->direction) were applied.

Accordingly, a layout algorithm that performs the layout operation of the example is specified by the classifier in Figure 4.

|                      |
|----------------------|
| Layout:Moving Vertex |
| vertex:any           |
| port:any             |
| position             |
| free = s             |
| side = s             |
| fixed = s            |
| edge:any             |
| structure            |
| two port = s         |
| hyper = s            |
| one port = n         |
| routing              |
| straight = n         |
| orthogonal = m       |
| channel = n          |
| bus = s              |
| arcs = n             |
| k-linear = n         |
| polyline = n         |

**Figure 4: Classifier for the Layout Algorithm**  
*Moving Vertex*

|  |
|--|
| GLML:Process Model                       |
| vertex:part                              |
| port:out                                 |
| position                                 |
| fixed = m                                |
| direction                                |
| directed                                 |
| output = m                               |
| vertex:joining step (not expanded)       |
| port:in                                  |
| position                                 |
| fixed = m                                |
| direction                                |
| directed                                 |
| input = m                                |
| port:out                                 |
| position                                 |
| fixed = m                                |
| direction                                |
| directed                                 |
| output = m                               |
| vertex:joining step (expanded)           |
| port:in                                  |
| position                                 |
| side = m                                 |
| east = m                                 |
| west = f                                 |
| north = f                                |
| south = f                                |
| direction                                |
| directed                                 |
| input = m                                |
| port:out                                 |
| position                                 |
| fixed = m                                |
| direction                                |
| directed                                 |
| output = m                               |
| edge:part to joining step (not expanded) |
| structure                                |
| hyper = m                                |
| arbitrary = m                            |
| routing                                  |
| orthogonal = m                           |
| bus = m                                  |
| direction                                |
| directed = m                             |

**Figure 5: Classifier for the GLML Process Model**

The output of the mapping operation is shown in Table 3.<sup>10</sup> The result is that the layout algorithm, as classified in Figure 4, and the GLML, as classified in Figure 5, can be mapped, so the layout is applicable to the language.

<sup>10</sup> For clarity, Table 3 does not show the full path of each feature, but only the last two elements in the abbreviated form: *feature.subfeature*.

**Table 3: Mapping result – Process Model and Moving Vertex**

| #               |                    | Process Model | Moving Vertex | ≡    |
|-----------------|--------------------|---------------|---------------|------|
| port classifier |                    |               |               |      |
| 1               | position.free      | any           | s             | true |
| 2               | position.side      | any           | s             | true |
| 3               | position.fixed     | any           | s             | true |
| 4               | directed.output    | any           | ∅             | true |
| 5               | directed.input     | any           | ∅             | true |
| 6               | side.east          | any           | ∅             | true |
| 7               | side.west          | any           | ∅             | true |
| 8               | side.north         | any           | ∅             | true |
| 9               | side.south         | any           | ∅             | true |
| edge classifier |                    |               |               |      |
| 10              | structure.two port | any           | s             | true |
| 11              | structure.hyper    | any           | s             | true |
| 12              | structure.one port | f             | n             | true |
| 13              | hyper.arbitrary    | any           | ∅             | true |
| 14              | routing.straight   | ∅             | n             | true |
| 15              | routing.orthogonal | m             | m             | true |
| 16              | routing.arcs       | ∅             | n             | true |
| 17              | routing.k-linear   | ∅             | n             | true |
| 18              | routing.polyline   | ∅             | n             | true |
| 19              | orthogonal.bus     | any           | s             | true |
| 20              | orthogonal.channel | ∅             | n             | true |
| 21              | direction.directed | any           | ∅             | true |

As another example for the mapping process we have classified one of the layout algorithms provided in the Eclipse Layout Kernel (ELK) [21]. Figure 6 shows an excerpt of the classifier for the *Graphviz Dot* algorithm.

|                         |
|-------------------------|
| Layout:ELK Graphviz Dot |
| vertex:any              |
| port:any                |
| position                |
| free = s                |
| side = n                |
| fixed = s               |
| edge:any                |
| structure               |
| two port = s            |
| hyper = n               |
| one port = n            |
| routing                 |
| straight = n            |
| orthogonal = s          |
| channel = n             |
| bus = n                 |
| arcs = s                |
| k-linear = n            |
| polyline = s            |

**Figure 6: Classifier for the Layout Algorithm *Graphviz Dot***

The classification of existing layout algorithms requires some expert knowledge. The documentation is a typical starting point, but not sufficient for this task. Test implementations or layout results have to be examined. Then again, classification has to be performed only once, and the resulting classifier can be used for multiple GLML. Having a library of already classified layout algorithms allows tool developers to create GLML editors quickly and easily.

Note that we have excluded some features of the layout algorithm for the sake of a simple comparison. The focus is on the port and edge classifier, which disregards additional constraints concerning the placement of vertices.

Mapping between the GLML and the layout is again a simple application of the mapping operator. To keep the example short we only show the differences in the mapping of *Graphviz Dot* compared to *Moving Vertex* in Table 4.

**Table 4: Mapping result – Process Model and *Graphviz Dot* (showing only the differences to Table 3)**

| #  |                    | Process Model | Graphviz Dot | ≡     |
|----|--------------------|---------------|--------------|-------|
| 2  | position.side      | m             | n            | false |
| 11 | structure.hyper    | m             | n            | false |
| 15 | routing.orthogonal | m             | s            | true  |
| 16 | routing.arcs       | ∅             | s            | true  |
| 18 | routing.polyline   | ∅             | s            | true  |
| 19 | orthogonal.bus     | m             | n            | false |

The overall mapping operator result shows that *Graphviz Dot* is not an applicable layout for the GLML. The mapping fails mainly because of the port handling.

Fully classifying the various layout algorithms in the ELK will show that there is at least one applicable algorithm. Applying the mapping operator to *ProcessModel* and *ELK Layered* results in a positive match.

## 7 Related Work

A classification scheme for the concrete syntax of GLML was presented in [25]. In this paper, that classifier serves as the basis for the classification scheme for the layout algorithms. In [25], layout algorithms were not classified. For the mapping between layout procedures and concrete syntax of a GLML and in order to answer the questions posed at the beginning, both classifiers based on the same feature model must exist. The authors are not aware of such classifiers and mapping procedures. The proximity of GLML to graphs and the requirements to develop layout procedures for the languages suggest a consideration of classifications in the graph drawing environment. A great number of works on graph drawing originated in the 1990s. As an example, we mention the works of Di Battista et al [18, 26]. Graph drawing methods exist for different classes of graphs. In [18] 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 [26] 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 [27].

The development of graphical modeling languages, such as UML and SysML, in the late 1990s and early 2000s and the associated development of software tools for model-driven software and systems engineering and development led to the growing importance of graph drawing algorithms in software development. In this context, for example, in the work of Sugiyama [28], a classification of graphs (trees, directed graphs, undirected graphs, compound graphs) was again made. In addition, he also classified placement conventions for vertices (free, parallel in line, concentric circles, radial line, orthogonal grid and grid based), routing conventions for edges (straight line, polyline, curved), and various drawing rules.

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. In [29], netlike schematics is the term used to differentiate them from graphs. Graphs are considered there, besides electrical diagrams, technological layouts, and petri nets as examples of netlike schematics. A general language called schematic structure description language (SSDL) is presented.

Metamodels corresponding to these netlike schematics are currently integrated in some frameworks [19–22] and implemented in layout procedures. 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.

So-called language workbenches coming from model-driven development (MDD) are to be considered. In language workbenches, the abstract and the concrete syntax and the mapping between them are defined as a metamodel (Eclipse GMF, EMF, GEF, Obeo, Sirius, and Microsoft DSL, MetaEdit), and graphical editors for DSL are generated from it. This is exactly where the challenges arise with regard to the development of suitable layout algorithms, which are currently not always well solved for practical applications [30]. In particular, the process of choosing a valid layout algorithm relies on expert knowledge or trial-and-error.

The presented classification scheme distinguishes vertices, ports, and edges as the most important model elements. A classification of layout methods based on a metamodel with ports

is given in [31]. The elements of the language and the so-called layout space are also classified there. The vertices (components) are subdivided, for example, according to their geometric shape (point, plane, spatial) and orientation (fixed, variable), but also according to whether their shape and size are fixed or variable. The layout space represents the area into which the graphical language is to be embedded. The classification of the layout space is also done with respect to the spatial dimension and according to the question of whether the layout space is discrete (grid) or continuous.

## 8 Conclusion and Future Work

In this paper, we aimed to improve the language engineering of GLML by taking human factors into account. In order to consider human factors in the application of GLML, layout procedures were identified as an essential component for tool development. The main hindrance for the integration of good layout procedures into modeling tools is the applicability of existing graph drawing algorithms.

We present an approach that provides the mapping of layout algorithms to GLML based on a general metamodel for GLML. For this purpose, we developed a mapping operator that builds on corresponding classification schemes for the concrete syntax of GLML and for layout algorithms. We demonstrated the performance of the mapping using a real-world example consisting of a graphical language and interactive layout support.

Usage of the mapping approach can lead to better modeling tools for newly developed languages. Frameworks for language engineering can implement the mapping to support GLML with unique concrete syntax by offering a well-fitted layout. This would enable a wider user base for both modeling languages and layout procedures.

The classification scheme will be extended in the future. We investigate qualitative values of features, e.g., a *preferred* value for features of GLML concrete syntax, or quantitative features, like the number of edges connectable to a single port.

The next development step will be the integration of the proposed mapping approach into an existing software framework [19]. This framework already supports the quick development of modeling tools for new GLML and a library of layout algorithms. A current research project aims to create new visualisations and editors assisting mechanical engineers during the design process of complex and interconnected structures in CAD models [15]. Therefore new GDSDL had to be designed. By mapping the languages onto layout algorithms, the development time for software tools can be shortened.

## 9 Acknowledgements

The authors thank the German Research Foundation (DFG) for the financial support of the research project “Method for the Model-Driven Design of Deep Drawing Tools” (project number BA 6300/1-3).



## REFERENCE

- [1] M. Brambilla, J. Cabot, and M. Wimmer, "Model-Driven Software Engineering in Practice," *Synthesis Lectures on Software Engineering*, vol. 1, no. 1, pp. 1–182, 2012, doi: 10.2200/S00441ED1V01Y201208SWE001.
- [2] B. Rumpe, "Modelling for and of Digital Twins," Oct. 12 2021. Accessed: ModDiT'21: 1st International Workshop on Model-Driven Engineering of Digital Twins. [Online]. Available: <https://gemoc.org/events/moddit2021>
- [3] OMG UML 2.5.1, *Unified Modeling Language*, v2.5.1. [Online]. Available: <https://www.omg.org/spec/UML/2.5.1/PDF>
- [4] OMG SysML 1.6, *Systems Modeling Language v1.6*. [Online]. Available: <https://www.omg.org/spec/SysML/1.6/>
- [5] B. Selic and S. Gérard, *Modeling and analysis of real-time and embedded systems with UML and MARTE: Developing cyber-physical systems*. Amsterdam: Elsevier Morgan Kaufmann, 2014. [Online]. Available: <http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10793779>
- [6] OMG BPMN 2.0.2, *Business Process Model and Notation*. [Online]. Available: <https://www.omg.org/spec/BPMN/>
- [7] Simulink - Simulation und Model-Based Design. [Online]. Available: <https://de.mathworks.com/products/simulink.html> (accessed: Jul. 23 2021).
- [8] G. Wrobel, R. Scheffler, and T. Kehrer, "Rethinking the Traditional Design of Meta-Models: Layout Matters for the Graphical Modeling of Technical Systems," in *2021 ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 351–360.
- [9] *Electrical and Electronics Diagrams*, Y14.15-1966, USAS Y14.15-1966, New York, 1966.
- [10] *Graphical symbols for diagrams: Part 2: Symbol elements, qualifying symbols and other symbols havin general application*, EN 60617-2, EN 60617-2, 1997.
- [11] M. Taylor and P. Rodgers, "Applying Graphical Design Techniques to Graph Visualisation," in *Proceedings / Ninth International Conference on Information Visualisation, 2005: 06 - 08 July 2005, [London, England, London, England, 2005*, pp. 651–656.
- [12] S. Sendall and W. Kozaczynski, "Model transformation: the heart and soul of model-driven software development," *IEEE Softw.*, vol. 20, no. 5, pp. 42–45, 2003, doi: 10.1109/MS.2003.1231150.
- [13] E. Augenstein, S. Herbergs, and G. Wrobel, "TOP-Energy - Ein Framework für Softwarelösungen in der Energietechnik," in *eOrganisation : Service-, Prozess-, Market-Engineering : 8. Internationale Tagung Wirtschaftsinformatik*, 2007, pp. 947–954.
- [14] CFaI Gesellschaft zur Förderung angewandter Informatik e. V., *TOP-Energy 3.0*. [Online]. Available: [www.top-energy.de/en/our-offer/top-energy-30](http://www.top-energy.de/en/our-offer/top-energy-30)
- [15] R. Scheffler, S. Koch, G. Wrobel, M. Pleßow, C. Buse, and B.-A. Behrens, "Modelling CAD Models: Method for the Model Driven Design of CAD Models for Deep Drawing Tools," in *4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, 2016, pp. 377–383.
- [16] P. Eades, W. Lai, K. Misue, and K. Sugiyama, "Preserving the mental map of a diagram," in *Proceedings of Compugraphics '91*, 1991, pp. 24–33.
- [17] D. Archambault and H. C. Purchase, "The mental map and memorability in dynamic graphs," in *IEEE Pacific Visualization Symposium (PacificVis), 2012: Feb. 28 2012 - March 2 2012, Songdo, Korea, Songdo, Korea (South)*, 2012, pp. 89–96.
- [18] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph drawing: Algorithms for the visualization of graphs*. Upper Saddle River, NJ: Prentice Hall, 1999.
- [19] G. Wrobel, R.-E. Ebert, and M. Pleßow, "Graph-Based Engineering Systems - A Family of Software Applications and their Underlying Framework," vol. 6, 2007, doi: 10.14279/tuj.eceasst.6.50.
- [20] J. Barzdins and A. Kalnins, "Metamodel Specialization for Graphical Language and Editor Definition," *BJMC*, vol. 4, no. 4, pp. 910–933, 2016, doi: 10.22364/bjmc.2016.4.4.20.
- [21] Eclipse Foundation, *Eclipse Layout Kernel: Graph Data Structure*. [Online]. Available: <https://www.eclipse.org/elk/> (accessed: Oct. 31 2021).
- [22] yWorks GmbH, *The Graph Model*. [Online]. Available: <https://docs.yworks.com/yfileshtml/#/dguide/graph>
- [23] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA CMU/SEI-90-TR-021, 1990. [Online]. Available: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231>
- [24] K. Czarnecki and C. H. P. Kim, Eds., *Cardinality-based feature modeling and constraints: A progress report*, 2005.
- [25] G. Wrobel and R. Scheffler, "Classification Scheme for the Concrete Syntax of Graph-like Modeling Languages for Layout Algorithm Reuse," in 2022, pp. 344–351.
- [26] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Computational Geometry*, vol. 4, no. 5, pp. 235–282, 1994, doi: 10.1016/0925-7721(94)00014-x.
- [27] C. D. Schulze, M. Spönemann, and R. von Hanxleden, "Drawing layered graphs with port constraints," *Journal of Visual Languages & Computing*, vol. 25, no. 2, pp. 89–106, 2014, doi: 10.1016/j.jvlc.2013.11.005.
- [28] K. Sugiyama, *Graph Drawing and Applications for Software and Knowledge Engineers*. Singapore: World Scientific Publishing Co Pte Ltd, 2002. [Online]. Available: <https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=1679597>
- [29] M. Pleßow and P. L. Simeonov, "Netlike Schematics and their Structure Description," *Workshop on Informatics in Industrial Automation*, pp. 144–163, 1989.
- [30] J. Cooper et al., "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.
- [31] M. May and W. Nehrllich, "Zu einigen Problemen der Layout-Synthese: Teil I: Platzierung," in *AdW/ZKI, Layout-Entwurf: Mathematische Probleme und Verfahren*, M. May, W. Nehrllich, and M. Weese, Eds., Berlin: Zentralinst. für Kybernetik und Informationsprozesse der Akad. der Wiss. der DDR, 1988, pp. 8–71.