Hindawi
Scientific Programming
Volume 2018, Article ID 7264060, 26 pages
https://doi.org/10.1155/2018/7264060

Research Article

Graph Drawing and Analysis Library and Its Domain-Specific Language for Graphs' Layout Specifications

Renata Vaderna, Željko Vuković, Igor Dejanović, and Gordana Milosavljević

Faculty of Technical Sciences, University of Novi Sad, Trg Dositeja Obradovića 6, Novi Sad, Serbia

Correspondence should be addressed to Gordana Milosavljević; grist@uns.ac.rs

Received 15 April 2017; Revised 9 September 2017; Accepted 19 February 2018; Published 1 April 2018

Academic Editor: Basilio B. Fraguela

Copyright © 2018 Renata Vaderna et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

This paper presents a graph drawing and analysis library written in Java called GRAD and its domain-specific language for simplifying the process of laying out graphs. One of GRAD's main goals is to provide completely automated ways of selecting and configuring a drawing algorithm, based either on the properties of a graph or on a user's input conforming to the domain-specific language. In order to verify the quality of GRAD's main features a user study was conducted. The participants were asked to grade diagrams visualized and laid out using different modeling tools, including one relying on GRAD, which received the best overall scores.

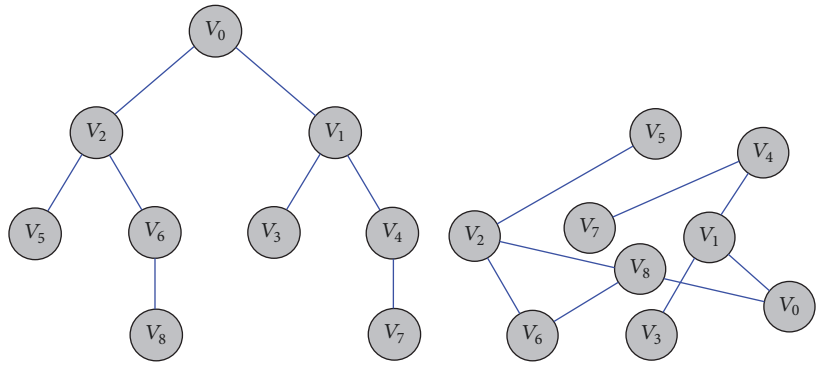
1. Introduction

Every graph, informally defined as a set of vertices and edges between them, can be drawn in a number of different ways [1]. Strictly theoretically speaking, it is only important that each vertex is mapped to a point on the plane and each edge to a curve between the appropriate two vertices. However, the arrangement of these elements directly impacts the graph's readability, understandability, and usability [1–5], that is, how clearly a viewer can understand the visualized information. An example of two different drawings of the same abstract graph is shown in Figure 1. Relationships between vertices of the graph are, for example, more evident in Figure 1(a).

Every diagram consisting of connected elements can be seen as a graph, UML class, activity, use case, business processes, and so on. In addition to conforming to some formal notation, diagrams can be created in accordance with their secondary notation. Secondary notation is defined as a set of visual cues which are not a part of a formal one [6]. In the graphical context, these cues are used to improve the readability of a formal notation and include the color of certain elements of the diagram and its layout. The study conducted by Schrepfer et al. discusses the impact of the secondary notation on the level of understanding of a business process model. It singles out the layout as the most important factor, determining how well both novices and experts perform while analyzing such models [7]. Similarly, the study by Purchase et al. focuses on the importance of different layout aesthetics in the domain of UML diagrams [8].

Bear the mentioned in mind, it is not surprising that many modeling tools (MagicDraw (http://www.nomagic.com/products/magicdraw.html), PowerDesigner (http://powerdesigner.de/en/), and Papyrus (https://eclipse.org/papyrus/)) and graphical modeling workbenches (Graphical Modeling Framework (GMF) (http://www.eclipse.org/modeling/gmp/), Sirius (http://www.eclipse.org/sirius/)) strive to offer the possibility of automatically laying out created diagrams in an aesthetically pleasing way. Such feature is particularly important in situations when a model created with some other tool is imported and visualized in the given one. In those cases, graphical elements are created and, preferably, laid out automatically. Furthermore, GMF and Sirius provide a way of expanding the set of available layout algorithms with new ones [9].

Developers of various new graphical editors as well as users of the existing ones looking to enhance their layout feature could, therefore, be interested in implementing one or more graph drawing algorithms. While there are types of layout techniques whose comprehension is not overly challenging, namely, circular, tree, and force-directed [10], implementing more sophisticated ones requires a significant



(a) More readable drawing (b) Less readable drawing

Figure 1: Two drawings of the same abstract graph.

knowledge of graph theory. The reason lays in the fact that such approaches often rely on complex graph analysis algorithms, such as decomposition of a graph into k-connected components, planarity testing and planar embedding, and finding graph's automorphisms [11–14]. For that reason, most developers are forced to use existing implementations of analysis algorithms, thus easing their own development of graph drawing algorithms or of already implemented layout algorithms.

There are several existing libraries focusing on the field of graph drawing. In this paper, however, emphasis is put on open-source solutions written in Java. Three such libraries can be singled out, due to offering the greatest number of stable implementations of layout algorithms: JUNG framework (http://jung.sourceforge.net), JGraphX (https://github.com/jgraph/jgraphx), and prefuse (http://prefuse.org). While all of them provide at least one implementation of a complex layout algorithm which can be applied on any graph, there are still some notable classes of such algorithms the libraries do not cover: straight-line, orthogonal, and symmetric above all. It can also be noted that none of the libraries have an impressive collection of analysis algorithms, only offering relatively basic ones, like depth-first search and Dijkstra's shortest path [16]. The mentioned shortcomings of available solution inspired the development of our new open-source graph analysis and drawing library (GRAD) (https://www.gradlibrary.net).

On top of offering a wide arrange of graph analysis algorithms, as well as implementations of layout algorithms belonging to classes omitted by other Java libraries, GRAD provides

(i) an easy way of calling its layout algorithms and retrieving the results,

(ii) two means of automatic selection and configuration of an appropriate layout algorithm, neither of which is supported by the other solutions:

(1) based on the properties of the graph,

(2) in accordance with the user's desires expressed through a domain-specific language.

Automation of the process of selection of an algorithm was motivated by the fact that not every user of a graphical editor can be expected to have a prior knowledge of graph drawing theory. Without it, a user would not know what to expect when calling a certain algorithm and how to configure its parameters to get the desired result. The first method analyzes the graph and singles out an algorithm based on the existence or absence of certain properties (like planarity, if the graph is a tree or not, etc.). However, a drawing produced as a result of application of a mathematically determined algorithm might not be in accordance with a user's personal preferences. This was the inspiration behind the development of GRAD's second automatic selection option. Using a domain-specific language (a computer language specialized to a particular domain [17]), the users can specify aesthetic criteria that the drawing should conform to. It should be mentioned that since the mid-1990s, several studies trying to measure the effectiveness of different aesthetic criteria have been conducted [3, 18–21]. However, with most of them focusing on some specific types of graphs, there is still much work to be done in this field. This fact also influenced the decision to leave the users the choice of aesthetic criteria, as opposed to automatically giving priority to one over the others.

There are two main reasons why a domain-specific language (DSL) would be the most suitable solution to the problem of letting the users choose the aesthetic criteria:

(1) GRAD is a library meant to be used by graph (diagram) editors. Unlike a graphical configuration tool, the language can be used by any project being developed in Java, regardless of which framework it uses, if it is a web or a desktop application and so on.

(2) DSLs can support more complex specifications, like the usage of the logical and/or/not operators.

In order to verify that the diagrams automatically laid out by GRAD are aesthetically pleasing and that the library is simple to use, a user study was conducted. The participants were software engineers who were firstly tasked with grading diagrams laid out using GRAD, as well as those whose elements were positioned by commercial tools

(PowerDesigner and MagicDraw) and a well-known open-source Papyrus tool. The participants were also asked to evaluate the intuitiveness of GRAD's code samples, as well as those of three other libraries. The goal of the study was to confirm that

(1) GRAD's capabilities of automatically laying out diagrams are better than of industry-leading tools,

(2) it is easier to write code for configuring and executing a layout algorithm when using GRAD than when using a different graph analysis and drawing Java library.

Certain aspects of the GRAD library have already been published [22–24]. Previous work focuses on the overview of the implemented layout algorithms, integration with existing graphical editors while also presenting difficulties faced when using other libraries, and challenges of laying out UML diagrams in particular. This paper will, therefore, put emphasis on the process of automatically choosing a suitable algorithm and the domain-specific language. Furthermore, an example of the library's usage and the evaluation of some of its most important features will also be presented.

The rest of the paper is structured as follows. Section 2 gives an overview of the basic graph theory definitions. Section 3 lists some other libraries and DSLs dealing with graphs. It also presents the layout aesthetic criteria and mentions the most popular classes of layout algorithms. Section 4 presents the GRAD library and the DSL, while Section 5 shows an example of the library's usage and describes the conducted user study and its results in more detail. Finally, Section 6 concludes the paper and outlines future work.

2. Basic Graph Theory and Graph Drawing Concepts

A graph (V, E) is an ordered pair consisting of a finite set V of vertices and a finite set E of edges, that is, pairs (u, v) of vertices [25]. If each edge is an unordered (ordered) pair of vertices, the graph is undirected (directed). An edge (u, v) is a self-loop if $u = v$. A graph is simple if it does not contain neither more than one edge between the same two vertices (multiedges) nor self-loops.

A path is a sequence of distinct vertices, $v_1, v_2, \ldots, v_k$, with $k \geq 2$, together with the edges $(v_1, v_2), \ldots, (v_{k-1}, v_k)$. A cycle is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$, with $k \geq 2$, together with the edges $(v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, v_1)$. A graph is said to be connected if there is a path from any vertex to any other vertex in the graph. A biconnected graph is a connected graph which has no vertices whose removal would disconnected it. Generally, a graph is k-connected if a set of $k - 1$ vertices whose removal disconnects it does not exist. So, a connected graph is 1-connected, a biconnected graph is 2-connected, and so on [26]. Graphs which contain at least one cycle are called cyclic graphs, while the ones that do not are known as acyclic.

A tree is a connected acyclic graph having no more than one path between a pair of vertices. A rooted tree is a tree with one distinguished vertex called the root. A forest is a disjoint union of trees.

A drawing Γ of a graph G maps each vertex v to a distinct point $\Gamma(v)$ of the plane and each edge (u, v) to a simple open curve $\Gamma(u, v)$ with endpoints $\Gamma(u)$ and $\Gamma(v)$ [25]. A drawing is planar if no two distinct edges intersect except, possibly, at common endpoints. A graph is planar if it admits a planar drawing.

A bend along an edge e of Γ is a common point between two consecutive straight-line segments that form e. Formally, if every edge of Γ has at most b bends, Γ is a b-bend drawing of G. A 0-bend drawing is also called a straight-line drawing.

The crossing number of a graph is defined as the minimum possible number of edge crossings with which the graph can be drawn [27]. An edge crossing is a point on the plane where two edges intersect. Having this in mind, a planar graph can be defined as a graph whose crossing number is zero.

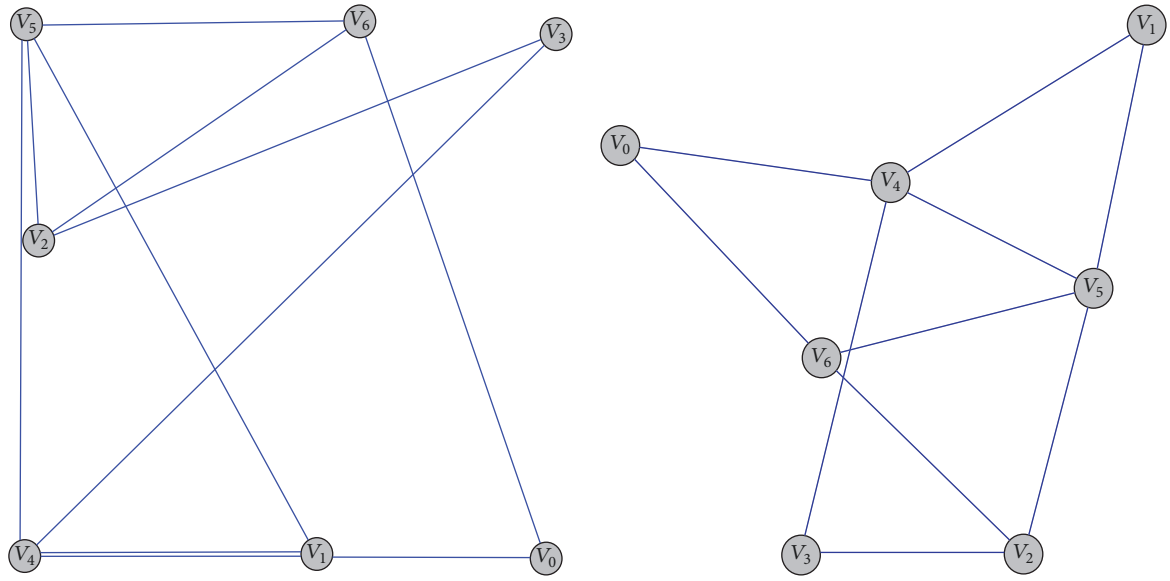
Finally, the process of automatically creating a drawing of a graph from the underlying graph structure is called automatic graph layout [28].

3. Related Work

In this section an overview of existing Java graph analysis and visualization libraries as well as domain-specific languages focusing on graphs will be given. Furthermore, the most important aesthetic criteria and classes of graph drawing algorithms will be presented.

3.1. Graph Drawing and Analysis Libraries and DSLs. There are quite a few Java libraries for graph analysis and visualization. Since our focus is on open-source solutions, JUNG framework, and JGraphX and prefuse can be singled out as the most notable libraries of the mentioned kind. All of them offer several complex graph layout algorithms, primarily focusing on tree drawing and force-directed methods, with JGraphX also offering a good implementation of a hierarchical algorithm. However, none of the libraries implement more complex graph analysis algorithms, which are needed for automatic detection of the suitable layout method. Additionally, they heavily focus on visualization, with their main goal revolving around generating fully functional graphical editors. This means that simply calling one of the graph layout algorithms they provide from a separately developed graphical editor is often too complicated. A detailed overview of the libraries and the difficulties of integrating them with existing editors can be found in [22] and [24], respectively. The rest of this section will therefore focus on presenting other domain-specific languages concerned with graphs, the most famous of which is Graphviz's (http://www.graphviz.org) DOT language [29].

DOT is a domain-specific language (DSL) for defining directed and undirected graphs. A description of a graph in the DOT language consists of naming all of the vertices it should contain and stating which of them are connected. The strength of this language lays in the fact that over 150 attributes of a graph and its elements can be customized: color, shape and label of each vertex and edge, the style of an arrowhead at an edge's end, the graph's margin, and font just to name a few. Among the numerous adjustable properties,



(a) A drawing created using Fruchterman-Reingold algorithm with randomly chosen parameters' values

(b) A drawing created using Fruchterman-Reingold algorithm with carefully chosen parameters' values

Figure 2: Two drawings of the same abstract graph created using Fruchterman-Reingold algorithm with different parameters' values.

there are also a few which focus on the graph's layout and characteristics of the resulting drawing. For example, it is possible to define how and if overlaps should be removed, the preferred lengths and representation of edges and the proximity of vertices. The edges can be drawn as splines, straight lines, or polylines.

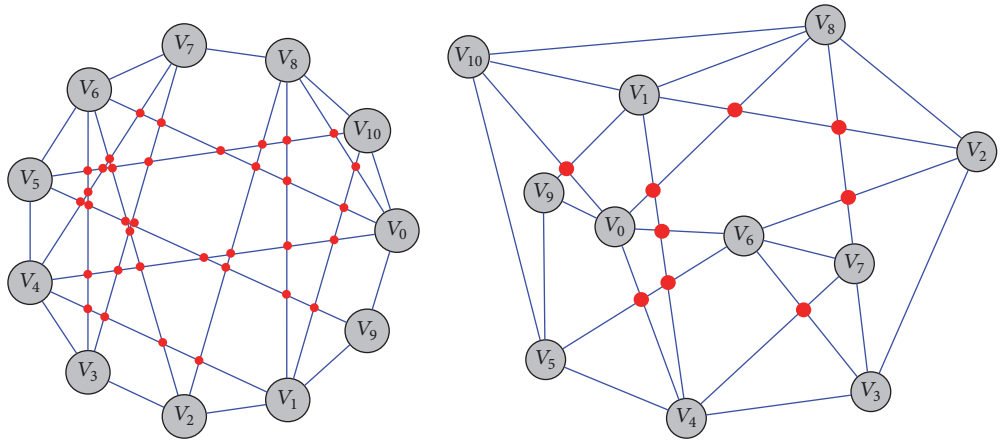
Once a graph has been described, one of Graphviz's layout tools needs to be invoked. Each of these tools is an implementation of one or more layout algorithms. Currently, Graphviz offers the following algorithms of the mention type: hierarchical, layered, spring, radial, and circular. Some of the parameters of these algorithms, such as the spring constant or the repulsive force used in an extended Fruchterman-Reingold force-directed model [30], are also a part of the DOT language and can be embedded in the graph's description.

Bear in mind that it can be noted that DOT is a sophisticated language for describing a structure of a graph and a variety of its properties, making it possible to visualize almost any type of diagram. However, the focus of this research is not creation of a graph and customization of the appearance of its element using a domain-specific language, but providing a way of specifying characteristics of its layout without knowing anything about graph drawing algorithms. DOT does not excel at this task. The first issue is the necessity to manually pick one of the available layout algorithms by calling the appropriate Graphviz's tool. This could cause a problem to an inexperienced user with no knowledge of graph drawing theory, who might not be familiar with spring, layered, and radial layout methods. He or she would probably struggle to determine which one to use in order to achieve the best result. Furthermore, while the DOT language does enable configuration of certain aspects of the layout process, that requires setting parameters of the chosen algorithm. Doing so properly, so that the resulting drawing is more aesthetically pleasing compared to one which would be generated using default parameters' values and can only be accomplished by the more knowledgeable users. For example, wrongly configuring a force-directed algorithm can lead to a drawing where the graph's vertices either are very far away from each other or are almost overlapping. A demonstration of the importance of proper configuration of the algorithms is shown in Figure 2, where the algorithm of Fruchterman and Reingold was applied twice, firstly with random values of the parameters (Figure 2(a)) and later with carefully chosen ones (Figure 2(b)).

GRAD's DSL, on the other hand, offers additional alternatives to specifying how a graph should be laid out and not just manual selection and configuration of an algorithm.

The DOT language is not the only DSL supporting specification of a graph's layout in some way. An example is a domain-specific language for visualizing software dependencies as graphs, called Graph [31]. In this context, graph vertices represent software elements, while edges are interpreted as dependencies between two entities. The Graph language is an internal DSL built in Pharo (http://pharo.org), which is a pure object-oriented language supported with an integrated development environment.

Graph allows its users to define vertices and edges of a graph, along with their properties such as shape and color. Furthermore, this DSL enables the definition of how a graph should be laid out and supports a number of well-known layout algorithms (force-directed, circular, and tree). It takes into consideration the fact that applying the same algorithm on all parts of a graph might not be the optimal approach. So, the DSL makes it possible to define a partitioning or a



(a) A drawing of a graph with a large number of edge crossings

(b) A drawing of a graph with a small number of edge crossings

Figure 3: Two drawings of the same abstract graph with different numbers of edge crossings.

subgraph to which a particular algorithm will be applied. Additionally, it allows selection of several layout algorithms to be applied in succession. However, it is always necessary to name and possibly configure the layout algorithms that will be used. There are no alternatives which a user not familiar with them could use.

Furthermore, there are a large number of projects for building and analyzing graphs that do not cover the problem of specifying how they should be laid out. One such example is a small language for simplifying the input of graph data in graph theory based Java programs [32]. It allows the users to specify vertices, edges, and their weights. This description is later meant to be used in implemented graph theory algorithms. Moreover, a domain-specific language called Green-Marl [33] can also be mentioned. This language was designed in order to ease implementation of graph analysis algorithms. It translates high-level algorithmic description into an efficient implementation in general purpose programming languages, for example, C++.

Summarily, most DSLs dealing with graphs in some way focus either on their complete definition and customization of various visual properties or on their analysis. To the best of our knowledge, there is no domain-specific language whose main purpose is to offer a more descriptive way of specifying how a graphs should be laid out and automatic selection, configuration, and execution of the appropriate algorithm. The first two mentioned DSLs touch upon the subject of a graph's layout. However, the users have to pick one of the available algorithms directly. In addition to this, their configuration is done by naming one or more parameters and providing the desired values. The other mentioned DSLs do not even cover the problem of laying out elements of a graph, as they focus on implementation of graph analysis algorithms. GRAD already contains such implementations and they are automatically used in the process of determining the most suitable layout algorithm.

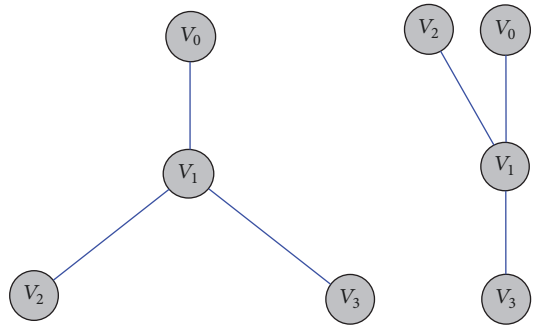
3.2. Graph Layout Algorithms and Aesthetics. A single graph can be drawn in a countless number of ways. Strictly theoretically speaking, all that is important is which vertices are connected. However, in practice the positions of a graph's vertices and edges within the drawing directly affect its readability and understandability [1–5]. So, a wide set of aesthetic principles believed to improve these aspects of a drawing has been proposed. Different designers of layout algorithms often focus on different criteria, believing that optimization of these measurable aesthetics produces nice graph drawings. This section presents the most common aesthetic criteria [3, 5, 28] and the most important classes of graph drawing algorithms [34].

3.2.1. Graph Aesthetics. The upcoming paragraphs describe the aesthetics which many graph drawing algorithms strive to achieve. These criteria are also at the core of the DSL presented in this paper, since they are intuitive on one hand and closely linked to layout algorithms on the other.

Minimization of the number of edge crossings is widely regarded as one of the most important aesthetic criteria. Readability of a two-dimensional graph layout is considered to be strongly dependent on this number. This was verified by several conducted studies [18, 28]. Bearing in mind that the main information given by an abstract graph is whether two vertices are connected by an edge, it is obvious that reducing the number of crossings significantly increases the readability [35].

In Figure 3 two drawings of the same graphs are shown. The drawing in Figure 3(a) has a large number of edge crossings (over 30), while the one in Figure 3(b) has notably less (only 9). The crossings are marked with red dots. This aesthetic criterion strongly favours drawing Figure 3(b), which is, undoubtedly, far more readable.

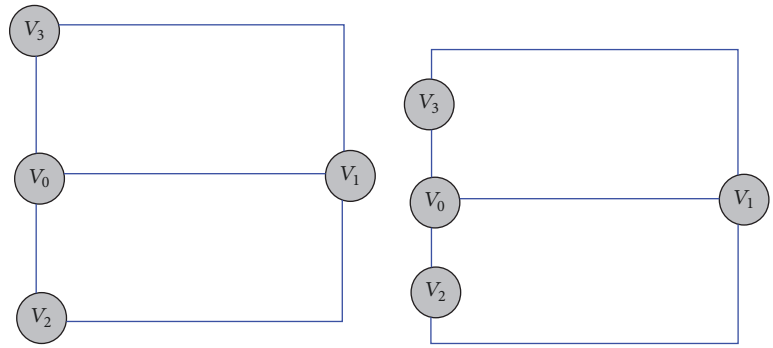
Maximization of minimum angles is an aesthetic criterion which states that the minimum angle between edges extending from a vertex should be maximized [19, 36]. Purchase et al. explain that the best possible result is achieved when all graph vertices have equal angles between all incident edges [37]. In Figure 4 two drawings of the same simple abstract graph are shown. The drawing in Figure 4(a) has maximized



(a) A drawing of a graph with maximized angles

(b) A drawing of a graph with a small minimum angle

Figure 4: Two drawings of the same abstract graph with significantly different values of the minimum angle between edges.



(a) A drawing of graph with a minimum number of bends

(b) A drawing of a graph with more bends than necessary

Figure 5: Two drawings of the same abstract graph, one with the minimum number of bends and one with a nonoptimal embedding.

angles at its center vertex, thus completely conforming to this aesthetic criterion. The other one, shown in Figure 4(b), has a small minimum angle.

Number of bends is a criterion which points out minimization of the number of bends as being important to the overall graph readability, especially in areas such as VLSI (very-large-scale integration) circuit layout, architectural design, and transportation problems [38]. Figure 5 contains two drawings of the same graph, where the one shown in Figure 5(a) has the minimum number of bends, whereas the one in Figure 5(b) has more bends than necessary, thus being less aesthetically pleasing according to the current criterion.

Uniform flow criterion names the flow of directed edges as something to pay attention to when creating a drawing of a graph. Generally, the direction of edges should be consistent [37]. A demonstration of this criterion is shown in Figure 6, consisting of two drawings of the same abstract graph. The drawing shown in Figure 6(a) has consistent flow and is more understandable than the drawing shown Figure 6(b), which has inconsistent flow.

Orthogonality aesthetic criterion claims that nodes and edges should be fixed to an orthogonal grid [38, 39]. So, the concept of orthogonality can be separated into two [37]:

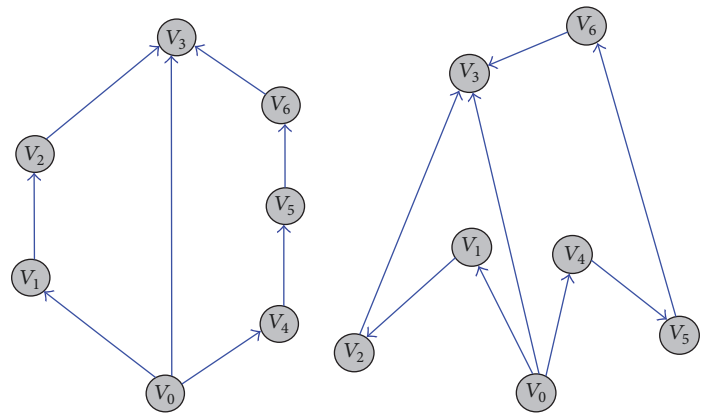
(i) edges and edge segments should follow the lines of an imaginary Cartesian grid;

(ii) vertices and bend points should make maximal use of an imaginary Cartesian grid.

In other words, segments of the edges should not deviate much from an orthogonal angle and vertices and bend points should be fixed to intersections on an imaginary unit grid, thus making maximal use of the grid area.

Symmetry is an aesthetic criterion that clearly reveals the structure and properties of a graph [12]. Knowing that every drawing of a graph has a trivial symmetry, it can be noted that this criterion enforces creating drawings of graphs with a nontrivial one, or, more ambitiously, with multiple symmetries. Figure 7 shows two drawings of the same graph, where the first one (shown in Figure 7(a)) has 8 nontrivial symmetries, and the other one (shown in Figure 7(b)) has only one. The drawing in Figure 7(a), on the other hand, also has 5 edge crossings, while the other drawing is planar. Still, most people prefer the drawing shown in Figure 7(a), which demonstrates importance of this criterion [40].

Symmetries of a drawing of a graph G are related to its automorphisms. An automorphism of a graph G is a



(a) A drawing of a graph with consistent flow

(b) A drawing of a graph with inconsistent flow

Figure 6: Two drawings of the same abstract graph, one with consistent flow and one with inconsistent.

mapping from its vertices back to those vertices such that the resulting graph is isomorphic with G [41]. An automorphism is geometric if there is a drawing which displays it.

In addition to previously described criteria, the authors of layout algorithms have also proposed the following ones [28]:

(i) Node distribution: nodes should be distributed evenly within a bounding box.

(ii) Edge lengths: edge lengths should not be neither too short nor too long.

(iii) Edge variation: edge lengths should be similar.

3.2.2. Graph Layout Algorithms. There are a large number of different graph drawing algorithms, with the oldest ones dating back to the 1950s. However, the field is still evolving, with new algorithms still being developed and old ones enhanced. These algorithms are often valued based on their computational efficiency and the extent to which they conform to one of more aesthetic criteria [28]. Some of the algorithms can be applied to every graph, while others require presence of certain features. This section will give an overview of the most popular classes of graph layout algorithms, with the emphasis being put on aesthetic criterion or criteria they focus on as well as the restrictions concerning their applications, if there are any.

Tree drawing algorithms are designed to produce nicely looking drawings of trees. These algorithms are often the best choice when hierarchical information should be conveyed. Tree drawing is among the best studied areas of graph drawing.

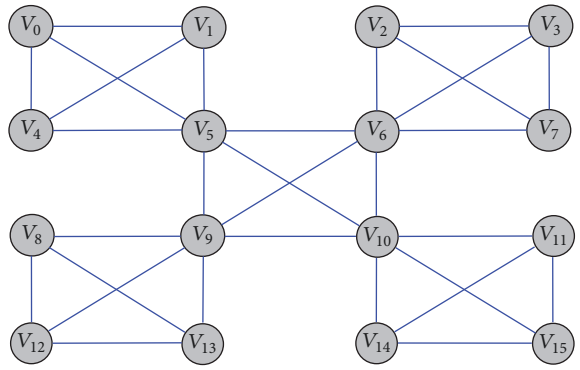
Some of the algorithms belonging to this class can only be used on binary trees (trees in which each vertex has at most two children) and others can be used on general trees as well. Depending on the algorithm of choice, the resulting tree can conform to a number of aesthetic criteria: planarity, orthogonality, symmetry, flow, similar lengths of edges, node distribution, and minimization of the number of bends. Furthermore, application of most of the available algorithms results in drawings which display more than one of the desirable properties; for example, they are planar and symmetric and have edges of similar lengths. A detailed overview of various approaches to drawing trees can be found in [42].

Straight-line drawing algorithms represent a class of graph drawing algorithms where edges can only contain straight-line segments. There are several types of these algorithms, which, in addition to focusing on the mentioned property also try to achieve goals set by one or more other aesthetic criteria. These include planar, polyline, convex, orthogonal, and rectangular drawings. Planar straight-line drawing algorithms rely on the fact that if a drawing can be drawn planar using edges of arbitrary shapes, they can also be drawn planar using just straight-line segments [11]. Moreover, planar straight-line drawing algorithms often accentuate the size of the angles and strive to produce convex, orthogonal, or polyline drawings. Convex drawings are defined as drawings where all faces are drawn as convex polygons. Orthogonal drawings only use horizontal and vertical line segments for edges and are, therefore, often quite visually pleasing. A more specific type of orthogonal drawings is rectangular drawings, which also make sure that each face is drawn as a rectangle [13]. It can easily be concluded that orthogonal drawings can only be constructed for graphs which do not contain a vertex with more than four edges entering and leaving it (its degree is four at most). Polyline drawings are more general and do not have this limitation. They directly focus on not allowing sizes of the angles to be smaller than some fixed threshold.

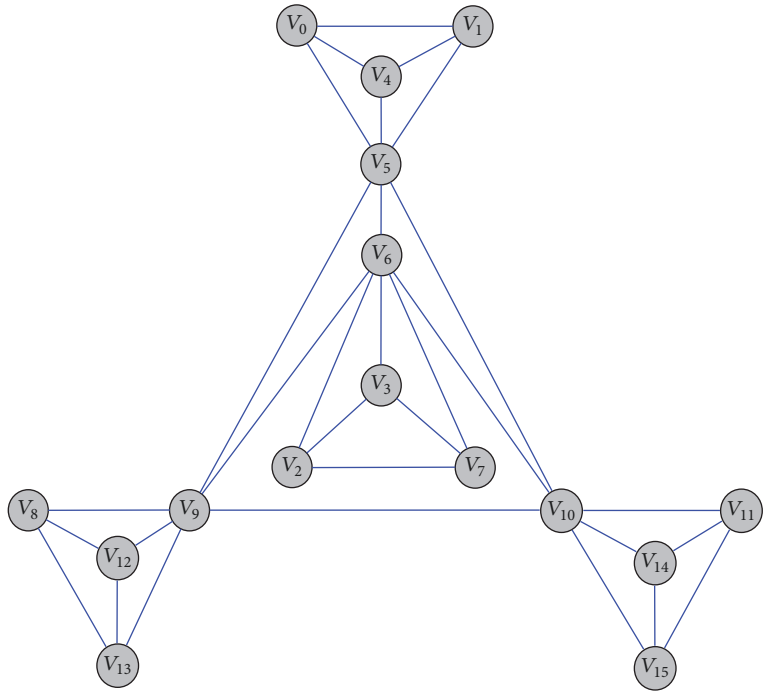
Hierarchical drawing algorithms can be used to draw directed graphs (or digraphs) which represent hierarchies [43]. Vertices represent entities and edges relationships between them. These algorithms produce drawings with consistent direction of edges, in accordance with one of the previously described aesthetic criteria.

Circular drawing algorithms are a class of graph drawing algorithms that partition the graph into clusters and place each node of each cluster onto the circumference of an embedding circle [44].

Symmetric graph drawing algorithms aim to draw a graph with nontrivial symmetry, or, more ambitiously, with as



(a) A nonplanar drawing of a graph with 8 symmetries



(b) A planar drawing of a graph with only an axial symmetry

Figure 7: Two drawings of the same abstract graph, with one having more symmetries and the other one being planar with not as many symmetries.

much symmetry as possible. In other words, these algorithms construct drawings of graphs with regard to the aesthetic criterion of the same name. Some of the algorithms require an automorphism of the graph to be passed and then proceed to construct a drawing which displays it. The more advanced ones, on the other hand, firstly find the largest automorphisms group or even the maximal planar group of this kind [12]. However, a need to manually specify an automorphism to be displayed could also arise, so both types of symmetric algorithms have their significance.

Force-directed algorithms are among the most important and most flexible graph drawing algorithms due to the fact that they can be used to lay out all simple undirected graphs. They only need the information contained within the structure of the graph itself [45]. Graphs drawn with these algorithms tend to be aesthetically pleasing, exhibit symmetries, and often produce crossing-free layouts for planar graphs. So, while they might not be able to guarantee that conditions set by the desired aesthetic criterion will be completely met, they can be used to produce drawings that satisfy a larger number of these criteria to some extent. For example, there is no guarantee that all drawings of planar graphs will necessarily be planar, but they will have a small number of edge crossings. There are many force-driven algorithms, with Tutte's 1963 barycentric method [46] being considered to be the first one. The most popular ones include the spring layout method of Eades [47], Kamada-Kawai [40], and Fruchterman-Reingold [30] methods.

4. GRAD and Its DSL for Graph Layout Description

This section presents our graph analysis and drawing library, GRAD, and describes its domain-specific language for

specifying graph layouts. The language is supposed to be used by both users knowledgeable in graph drawing theory and those who lack such experience, so it features several ways of defining how a graph should be laid out. The most complex of them enables full configuration of the layout process, while the simplest ones only require the users to name certain characteristics of the resulting drawing, based on which the best algorithm is automatically selected and configured. More precisely, one of the available algorithms whose resulting drawings exhibit the greatest number of the mentioned characteristics is chosen. Automatic detection of the preferable algorithm would not be possible without GRAD's implementations of graph analysis algorithms. Therefore, the GRAD library will be described in more detail before the DSL itself is presented.

4.1. Graph Drawing and Analysis Library (GRAD). GRAD is an open-source graph drawing and analysis library written in Java aiming to provide

(1) a large number of algorithms from graph and graph drawing theory,

(2) automatic selection, configuration, and execution of a layout algorithm based on the properties of a graph or a user's descriptive input,

(3) simple integration with existing graphical editors.

GRAD ports the best implementations of layout algorithms from other open-source graph drawing and analysis libraries, JUNG framework, JGraphX, and prefuse, and adds several original ones. GRAD's layout algorithms include both those which were specifically designed with a certain type of graphs in mind, for example, biconnected planar, and those which can be applied to any graph, generating drawings conforming to one or more aesthetic criteria. Overall, GRAD includes the following layout algorithms:

(i) several tree drawing, ranging from the standard level-based approaches, to those creating radial tree drawings,

(ii) the majority of the well-known force-directed algorithms (Kamada-Kawai, Fruchterman-Reingold, spring, organic [48], and a so-called ISOM layout based on Meyer's self-organizing graph methods [49]),

(iii) one hierarchical algorithm [50],

(iv) a symmetric layout algorithm, based on the work of Carr and Kocay [51],

(v) a straight-line drawing algorithm based on Tutte's theorem [46].

(vi) Chiba's convex straight-line drawing algorithm [52],

(vii) a circular drawing algorithm with the optional optimization of the number of edge crossings [53],

(viii) a simple layout algorithm which places a certain number of vertices in one row before continuing to the next one, called box layout.

Table 1: Times the algorithms need to lay out randomly generated graphs with 1000 vertices and 2000 edges.

Algorithm	Time [ms]
Spring	724
Fruchterman-Reingold	689
Kamada-Kawai	5232
ISOM	461
Fast-organic	9584
Organic	90273
Hierarchical	56100
Symmetric	523
Circular	419
Box	114

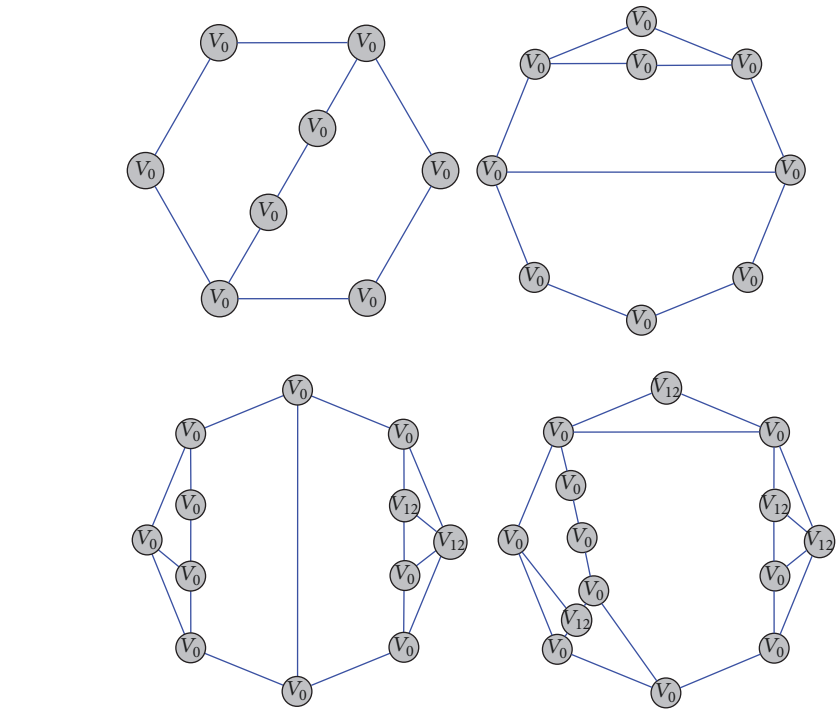
Table 2: Times the algorithms need to lay out trees with 1000 vertices.

Algorithm	Time [ms]
Level-based tree	60
Radial tree	52
Compact tree	140
Balloon tree	158
Node-link tree	461
Hierarchical	621

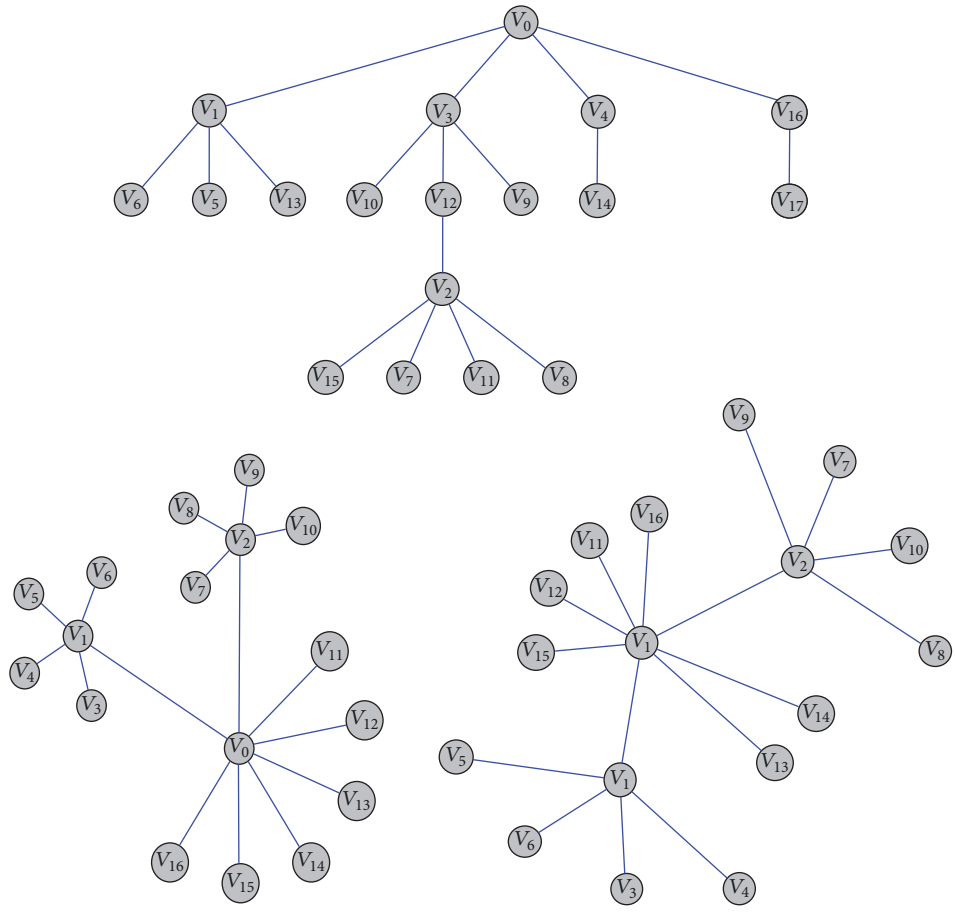
Furthermore, an orthogonal layout algorithm based on visibility representations [54] is currently in development. The last five algorithms are GRAD's original implementations. To the best of our knowledge, some of them (symmetric, Chiba's, optimized circular algorithms) have not been implemented in Java before. Some examples of drawings of graphs created using GRAD's layout algorithms are shown in Figure 8. More examples are available at [55] (GRAD's official site). A more detailed overview of the algorithms can be found in [22]. The examples were created using GRAD's simple graphical editor (https://github.com/renatav/GraphDrawing/tree/master/GraphEditor), developed in order to support familiarization and experimentation with the available algorithms.

In addition to being judged based on the extent to which they conform to one or more aesthetic criteria, layout algorithms are rated in accordance with their computational efficiency, that is, the quantity of resources they need in order to compute their results, positions of graph's elements. In Tables 1 and 2 times certain implementations which need to lay out bigger graphs are presented. The graphs used in the tests had 1000 vertices and twice as many edges. Algorithms only meant to be performed on smaller graphs, like Tutte's embedding, and which can only be applied to very specific graphs, like Chibba's algorithm, were omitted.

Hu developed highly effective force-directed algorithms and concluded that a graph of similar size to the ones used in the previously mentioned tests can be laid out in under 1 second [56]. Some of the algorithms offered by GRAD live up to that standard, while also producing understandable drawings. The ISOM algorithm is the best example. It is

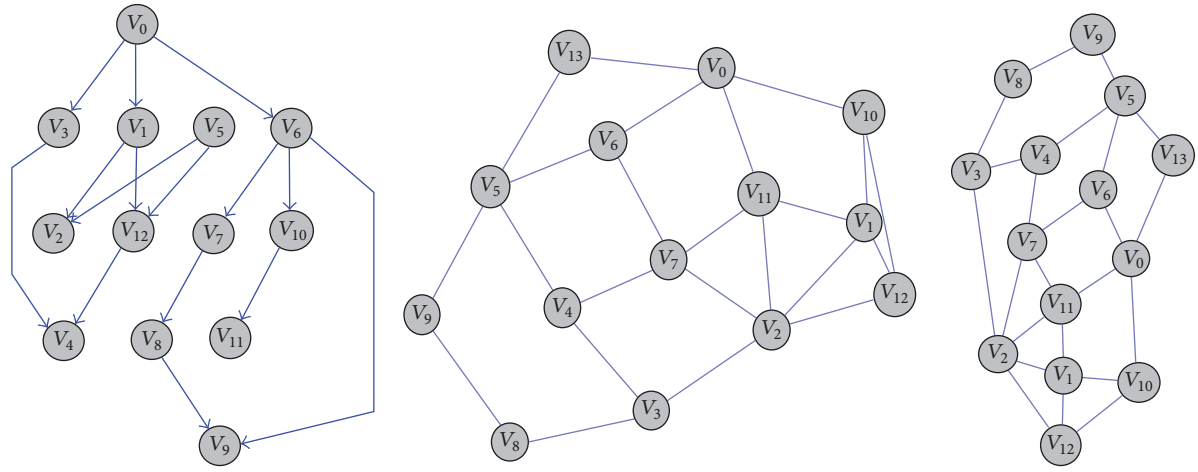


(a) Drawings generated using Chiba's convex algorithm



(b) Level-based, balloon, and radial tree drawings

Figure 8: Continued.



(c) A hierarchical drawing (d) Drawings generated using Kamada-Kawai and organic force-directed algorithms

FIGURE 8: Examples of drawing of graphs created using GRAD's layout algorithms.

also evident that all tree methods are very efficient. The hierarchical one performs poorly when used to lay out a general graph, while being much faster if the graph is actually a hierarchy.

Moreover, GRAD also performs some postprocessing, thus enhancing the results produced by the algorithms. Since very few of the algorithms route loops and multiple edges, GRAD detects them and recalculates their positions in order to avoid overlapping of edges and correctly show those which connect one vertex to itself. Additionally, some of the algorithms take into consideration the sizes of the vertices, but not all. Therefore, GRAD can also slightly move the vertices after an algorithm has been performed in order to prevent their overlapping. It should be mentioned that GRAD's original implementations take into account the sizes of a graph's vertices.

In addition to providing the mentioned layout algorithms, GRAD implements many concerned with graph analysis, traversal, decomposition, and so on. Most notable of them are the following:

(i) Dijkstra's shortest path [16],

(ii) checking connectivity, biconnectivity, and triconnectivity of a graph,

(iii) finding simple cycles of a directed or an undirected graph,

(iv) Fraysseix-Mendez [57], Boyer-Myrvold [58], and PQ-tree [59] planarity testing,

(v) finding a planar embedding of a graph,

(vi) Hopcroft-Tarjan division of graph into triconnected components [60], which is considered to be one of the most difficult algorithms to implement,

(vii) Fialko and Mutzel's 5/3 approximation algorithm for the planar augmentation problem [61],

(viii) McKay's canonical graph labeling algorithm (nauty) for finding permutations [62].

Graph analysis algorithms are of great significance to the field of graph drawing, with some of them being the core of different layout algorithms. For example, implementation of Chiba's convex drawing algorithm includes planarity and triconnectivity checking and splitting the graph into triconnected components, while the orthogonal algorithm based on visibility representations requires finding a planar embedding. GRAD's analysis algorithms can also be used by other developers interested in implementing additional layout algorithms.

Furthermore, the analysis algorithms can be used to check if a particular drawing algorithm can be applied to a given graph. Although there exist layout algorithms that can be applied to any type of graphs, using an algorithm which primarily focuses on graphs such as the given one is usually a far better solution. For example, if a graph is a tree, it should be laid out using a tree drawing algorithm and not using a hierarchical or force-directed one. This fact is the basis of the automatic detection of the best drawing algorithm and therefore of the implementation of some of the library's most important features. Additionally, algorithms like planar augmentation can transform a graph into a suitable one prior to execution of a layout algorithm. For instance, some edges can be added to make a graph biconnected, seeing that many algorithms require that property.

Finally, it can be mentioned that GRAD's algorithms can directly be used to lay out elements of any diagram. In fact, the ease of calling any of the library's algorithms from a separately developed graphical editor is one of its main goals. If that is possible, the editor's classes corresponding to graph vertices and edges should implement GRAD's `Vertex` and `Edge` interfaces, thus specifying information needed by the algorithms, sizes of the vertices and origins and destinations of the edges. If for whatever reason this requirement cannot be satisfied, GRAD provides two classes (`GraphVertex` and `GraphEdge`) which already implement the mentioned interfaces. If a GRAD's vertex is then created for each of the editor's vertices and a GRAD's edge for each of its edges, any

of our library's algorithms can be called. A more thorough explanation can be found in [23].

The library is currently being used in our Kroki mockup tool's (http://www.kroki-mde.net) lightweight UML diagram editor to lay out models sometimes containing over 600 classes.

4.2. Graph Layout DSL. The development of DSLs consists of the following phases: decision, analysis, design, implementation, testing, deployment, and maintenance [17, 63, 64]. This section will present a DSL for specifying how a graph should be laid out and describe each of its development phases.

4.2.1. Decision. The decision phase focuses on the question if developing a DSL is worthwhile or not. In other words, benefits of the new language can justify time and effort which needs to be put into its development. There are three main reasons why a positive answer was given to these questions in case of GRAD's DSL language:

(1) One of GRAD's main goals is to provide a generic way of its integration with any graphical editor. This can be accomplished by using a DSL, as discussed in Section 1. Additionally, the DSL can offer end-users of such an editor, who might be neither programmers nor knowledgeable in graph theory, the possibility of descriptively specifying how a diagram's layout should look like. That is, which features (e.g., orthogonality, symmetry, and uniform flow) should the layout exhibit and which are not desired. This option could save the mentioned users a lot of time and effort, as they would not have to experiment with configuring and calling different algorithms in order to lay out a diagram in accordance with their personal preferences. The developers of the graphical editors would only need to provide a regular input component, where the users would enter text conforming to the DSL.

(2) Although any of GRAD's layout algorithms can be configured and called directly by a developer using this library, a DSL can provide a way of achieving the same result in fewer lines of code (compare Listing 9 with 10 or Listing 11 with Listing 12). Additionally, studies showed that when using a DSL, developers are more accurate and more efficient in program comprehension than when using a GPL [65].

(3) Modern libraries and languages provide programmers with tools which make developing their own external DSL (DSL which is parsed independently of the host purpose language [66]) within their reach [67]. One such example is textX [68], a metalanguage for DSL specification, with whose help a new textual language can quickly be created.

4.2.2. Domain Analysis. Detailed domain analysis precedes design and implementation of a DSL [64]. Its goal is to define domain of focus, collect necessary domain information, and optionally create a domain model. More often than not, domain analysis is done informally [69], which was also the case during development of GRAD's DSL. The domains of particular interest to the mentioned language include graph drawing, which was introduced in Section 3.2, as well as GRAD's layout API, presented in [23] and GRAD's official site [55].

4.2.3. Design. Language design consists of the definition of constructs and language semantics [64]. Approaches to DSL design can be characterized along two orthogonal dimensions: the relationship between the DSL and existing languages and the formal nature of the design description [17].

The first dimension refers to construction of the language, either by basing it on an existing one or by creating it from the beginning. GRAD's DSL was built from scratch using textX. Although recent studies show that using an embedded (internal) DSL (one based on an existing language) can be made easier for nonprogramming users if the IDE (integrated development environment) is properly customized [70], external DSLs, which have no commonalities with existing languages (and therefore less syntactic noise), are generally regarded as more intuitive to the domain users. GRAD's DSL is an external domain-specific language, as it should also offer the option of being used by end-users of graphical editors looking to lay out their diagrams. Furthermore, many Java programmers prefer to configure certain elements of the applications they are developing through external files (e.g., JSON, YAML, and properties), pointing out code readability as the main advantage of this approach. An external DSL can significantly improve understandability of code for configuring and executing a layout algorithm, in addition to making it a lot shorter.

The second dimension incorporates formal or informal design of the language, that is, its syntax and semantics. Although building a DSL often seems intuitive, it is important to evaluate its usability systematically, and not just in the final stage of its development [71]. While no formal framework was used to achieve this, it can be noted that the members of our Kroki mockup tool's development team (http://www.kroki-mde.net/people/), who are also among the target end-users of this language, were involved in its design, by evaluating intuitiveness of the grammar and potential usefulness of different concepts.

Syntax specification of GRAD DSL's was done using the textX metalanguage, whose main features include the possibility of defining both language concrete syntax and its metamodel using a single description [68]. The description consists of rules, which are the basic building blocks of the textX metalanguage. The base rules of the DSL are shown in code Listing 1, while the complete specification can be found in [72]. The language was designed in such a way that the users could get the feeling of directly issuing orders to the application. So, the rules are meant to feel like sentences if written without skipping the optional words. If, however, a user is primarily interested in minimizing the length of the input, all parts of the rules followed by the `?` symbol can be omitted.

The `LayoutGraph` rule defines how to trigger the process of laying out the a graph. That is accomplished by typing the

```
LayoutGraph: 'lay' 'out' 'graph' LayoutEnum;
LayoutEnum: LayoutAlgorithm | LayoutStyle | AestheticCriteria |
    AestheticCriteriaMath;
LayoutStyle: 'using'? 'style' LayoutStyleEnum;
LayoutStyleEnum: 'automatic' | 'circular' | 'tree' | 'hierarchical' | '
    symmetric' | 'general';
AestheticCriteria: ('conforming' 'to')? 'criteria' (AestheticCriterion
    ',')+;
LayoutAlgorithm: 'using'? 'algorithm' LayoutAlgorithmEnum;
AestheticCriteion: EdgeCrossings | MinimumAngles | MinimumBands;
LayoutAlgorithmEnum: TreeAlgorithm | StraightLineAlgorithm |
    HierarchicalAlgorithm;
```

Listing 1: Base rules of the graph layout DSL.

```
MinimumBands: ('minimization' 'of')? 'edge' 'crossings';
```

Listing 2: An example of a rule corresponding to an aesthetic criterion.

words `lay out graph`, with any number of spaces between them, followed by the input conforming to one of the four available layout methods. Those include

(i) directly naming a layout algorithm and optionally setting its parameters,

(ii) naming the general style of the layout (hierarchy, circular, symmetric, etc.), including the possibility of stating that the layout process should be done automatically,

(iii) naming aesthetic criteria that the drawing should conform to (minimization of edge crossings, maximization of minimum angles, uniform flow, etc.) in the order of importance,

(iv) writing expressions using logical operators and aesthetic criteria, thus being able to state that, for example, a certain criterion should not be present in the resulting drawing, while two others both should and so on.

The rule `LayoutAlgorithm` corresponds to the firstly mentioned method of specifying how to lay out a graph. This rule references other ones, all of which represent a layout algorithm that can be used. If an algorithm has configurable parameters, they can be specified as well. This is, however, optional and it is assumed that the default values will be set if the user does not provide them. Not all algorithms were listed, due to their number.

The rule `LayoutStyle` represents the second method, which is the easiest one to use, as it is only necessary to select the overall style of the layout. `LayoutStyleEnum` represents a choice of these styles.

`AestheticCriterion` enables the third method of layout specification. It references other rules, where each of them corresponds to one of the aesthetic criteria described in Section 3.2.1. The repetition is achieved through usage of symbol +, which defines that one or more criteria must be listed. Since there are many such rules, only one will be presented, and it can be seen in code Listing 2.

In this rule, the sequence of two words, `minimization of`, is completely optional.

The most complex of the definitions is rule `AestheticCriteriaMath`, shown in code Listing 3. It introduces logical operators into the language, respecting their mathematical priorities. There is no limit to the length of the expression.

The choice to base this and the previous approach on the aesthetic criteria was inspired by the fact that they are descriptive and self-explanatory but also closely related to the layout algorithms. As mentioned before, these algorithms tend to strongly focus on producing drawings conforming to one or more aesthetic criteria.

Code Listing 4 shows a few examples of inputs conforming to GRAD's DSL incorporating everything previously mentioned.

The DSL also supports specification of how one or more subgraphs of the whole graph should be laid out. Since the rules through which that is accomplished are very similar to the ones shown in Listing 1, they were omitted from the overview of the grammar.

Semantics of a language can be specified formally, using attribute grammars, denotational semantics, rewrite systems, or abstract state machines, but also informally, usually using a natural language with optional program examples [17, 64]. GRAD's DSL is a simple language, with a very high level of abstraction, whose programs are transformed into complex Java code which relies on GRAD's layout API. As a result, the semantics of the language can most effectively be explained using the second mentioned method. Firstly, the meanings of the main rules of the language will be clarified via Table 3. Secondly, one example of how an input conforming to the

```
AestheticCriteriaMath: CriteriaExpression;
CriteriaFactor: 'not'? (AestheticCriteion | ('(' CriteriaExpression ')')
    );
CriteriaTerm: CriteriaFactor CriteriaAndFactor*;
CriteriaAndFactor: 'and' CriteriaFactor;
CriteriaExpression: CriteriaTerm CriteriaOrTerm*;
CriteriaOrTerm: 'or' CriteriaTerm;
```

Listing 3: Language rules for the support of logical operators.

```
lay out graph using style symmetric
lay out graph using algorithm Kamada-Kawai
lay out graph conforming to planarity, bends
lay out graph not(planarity and flow) or symmetry
```

Listing 4: Examples inputs conforming to the DSL.

```
radial tree(horizontal distance = 5, vertical distance = 20)
```

Listing 5: An input conforming to the `LayoutAlgorithm` rule.

```
//select algorithm
LayoutAlgorithms algorithm = LayoutAlgorithms.RADIAL_TREE;
//set properties
GraphLayoutProperties layoutProperties = new GraphLayoutProperties();
layoutProperties.setProperty(RadialTreeProperties.X_DISTANCE,5);
layoutProperties.setProperty(RadialTreeProperties.Y_DISTANCE,20);
//initialize the layouter, assuming that vertices and edges
//are lists containing elements of the graph
Layouter<GraphVertex, GraphEdge> layouter = new Layouter<>(
                            vertices, edges, algorithm, layoutProperties);
Drawing<GraphVertex, GraphEdge> drawing = layouter.layout();
```

Listing 6: JAVA code corresponding to the layout specification from Listing 5.

DSL is mapped onto GRAD's API calls will be given for each of the four layout specification methods.

As previously mentioned, inputs (programs) written in GRAD's DSL are transformed into Java code mostly consisting of this library's layout API calls. The upcoming code listings are pairs of DSL and resulting Java code samples. Code Listing 5 shows an example of input conforming to the `LayoutAlgorithm` rule, whereas the JAVA/GRAD code which should be executed is encapsulated in Listing 6. Similarly, code Listing 7 contains an example of layout specification written in accordance with the `LayoutStyle` rule, while Listing 8 shows the corresponding Java code. Finally, code Listings 9, 10, 11, and 12 are pairs of inputs written in the DSL, conforming to rules `AestheticCriteria` and `AestheticCriteriaMath`, respectively, and the resulting API calls.

```
lay out graph using style circular
```

Listing 7: An input conforming to the `LayoutStyle` rule.

4.2.4. Implementation. Implementation is a phase in construction of DSLs when the most suitable approach of DSL development is chosen and performed. These approaches include creation of interpreters, compilers, and preprocessors to name a few [17]. GRAD's DSL uses an interpreter, which relies on textX's ability to create an object graph (model) from the input string conforming to the DSL.

While processing the grammar, textX creates the metamodel of the language. More precisely, a class is created for

Table 3: DSL expressions and their semantics.

DSL input	Semantics
`lay out graph using algorithm AlgorithmName (property1 = value1, property2 = value2)`	Select algorithm named AlgorithmName and set value of its property called property1 to value1, property2 to value2. Execute the selected algorithm
`lay out graph using style StyleName`	Select the best algorithm automatically. If the provided style is not automatic, limit the set of considered algorithms to those belonging to the appropriate group. If the style is circular, hierarchical, tree or symmetric, select an algorithm belonging to the group of the same name. If it is general, select a force-directed algorithm. Execute the selected algorithm
`lay out graph conforming to criteria criterion1, criterion2, criterion3`	Select a layout algorithm which can be applied to the given graph and produces drawings which exhibit characteristics favored by aesthetic criterion criterion1 and, if possible, criterion2 and criterion3. Execute the selected algorithm
`lay out graph not?(criterion1 and criterion2) or not?(criterion3 and criterion4)`	Select a layout algorithm which satisfies either criterion1 and criterion2 (or doesn't satisfy at least one of them, depending on the presence or absence of the negation operator), or criterion3 and criterion4. Execute that algorithm

```
//select the best algorithm based on properties of the graph,
//but only consider those belonging to a class specified by style
LayoutAlgorithms algorithm =
                LayoutPicker.pickAlgorithm(graph, "circular");
GraphLayoutProperties layoutProperties = DefaultGraphLayoutProperties.
                getDefaultLayoutProperties(algorithm, graph);
Layouter<GraphVertex, GraphEdge> layouter = new Layouter<>(
                vertices, edges, algorithm, layoutProperties);
Drawing<GraphVertex, GraphEdge> drawing = layouter.layout();
```

Listing 8: JAVA code corresponding to the layout specification from Listing 8.

each rule of the grammar. This means that when a program written in the DSL is parsed and the model is created, each of the model's objects is an instance of its metamodel's class. A model of the input string from code Listing 13 generated by textX is shown in Figure 9.

Once the model is available, the interpreter performs its transformation into appropriate GRAD API calls. For example, names and properties of the algorithms are transformed into corresponding GRAD's `LayoutAlgorithms` enumeration values and entries in `GraphLayoutProperties` map. While implementing such transformations is not a complicated task and syntax and semantics of the language are fairly simple, its complexity is encapsulated in `PickAlgorithm` methods. They contain implementations of selection of an appropriate layout algorithm based on properties of the graph and the user's input. A detailed description of these procedures will be given in Section 4.3.

4.2.5. Testing and Deployment. Testing and deployment are the next phases of DSL development. As the name suggests, a DSL's evaluation is performed during the testing phase, while deployment marks the point in time when applications constructed with the DSL are used [64]. The GRAD library and its DSL were integrated with our Kroki mockup tool [73], whereas their evaluation was performed by conducting a user study. The study showed that more participants would prefer to use the DSL to trigger the layout process than any of the other options.

4.2.6. Maintenance. Maintenance is the last phase in DSL construction, which occurs when there are new requirements. When additional layout algorithms are implemented, the DSL should be expanded to support them. This means adding new grammar rules, implementing more transformations, and enhancing the procedure described in Section 4.3, to take newly added algorithms into considerations. Due to its ability to build both a metamodel and a parser from a single grammar description, as well as its dynamic nature, textX makes iterative development of new DSLs easy.

4.3. Integration of the DSL with GRAD. The main focus of the DSL is to trigger execution of the appropriate layout

```
lay out graph conforming to planarity, symmetry, flow
```

Listing 9: An input conforming to the LayoutCriteria rule.

```
List<AestheticCriteria> criteria = new ArrayList<AestheticCriteria>();
criteria.add(AestheticCriteria.PLANAR);
criteria.add(AestheticCriteria.SYMMETRIC);
criteria.add(AestheticCriteria.UNIFORM_FLOW);
LayoutAlgorithms algorithm = LayoutPicker.pickAlgorithm(graph, criteria)
    ;
GraphLayoutPropertieslayoutProperties = DefaultGraphLayoutProperties.
                getDefaultLayoutProperties(algorithm, graph);
Layouter<GraphVertex, GraphEdge> layouter = new Layouter<>(
                vertices, edges, algorithm, layoutProperties);
Drawing<GraphVertex, GraphEdge> drawing = layouter.layout();
```

Listing 10: JAVA code corresponding to the layout specification from Listing 9.

```
lay out graph not(symmetry and angle) or edge crossings
```

Listing 11: An input conforming to the AestheticCriteriaMath rule.

```
List<Pair<List<AestheticCriteria>, List<AestheticCriteria>>> orPairs =
        new ArraList<Pair<List<AestheticCriteria>,
        List<AestheticCriteria>>>();
List<AestheticCriteria> positiveCriteria =
        new ArrayList<AestheticCriteria>();
positiveCriteria.add(AestheticCriteria.MINIMAL_EDGE_CROESSES);
List<AestheticCriteria> negativeCriteria =
        new ArrayList<AestheticCriteria>();
negativeCriteria.add(AestheticCriteria.SYMMETRIC);
negativeCriteria.add(AestheticCriteria.MINIMUM_ANGLES);
orPairs.add(new Pair<List<AestheticCriteria>, List<AestheticCriteria>>(
                  positiveCriteria, negativeCriteria));
LayoutAlgorithms algorithm = LayoutPicker.pickAlgorithm(graph, orPairs);
GraphLayoutPropertieslayoutProperties = DefaultGraphLayoutProperties.
        getDefaultLayoutProperties(algorithm, graph);
Layouter<GraphVertex, GraphEdge> layouter = new Layouter<>(
        vertices, edges, algorithm, layoutProperties);
Drawing<GraphVertex, GraphEdge> drawing = layouter.layout();
```

Listing 12: JAVA code corresponding to the layout specification from Listing 11.

```
lay out graph not(symmetry and angle) or planarity or edge crossings
```

Listing 13: An input conforming to the DSL whose model will be shown.

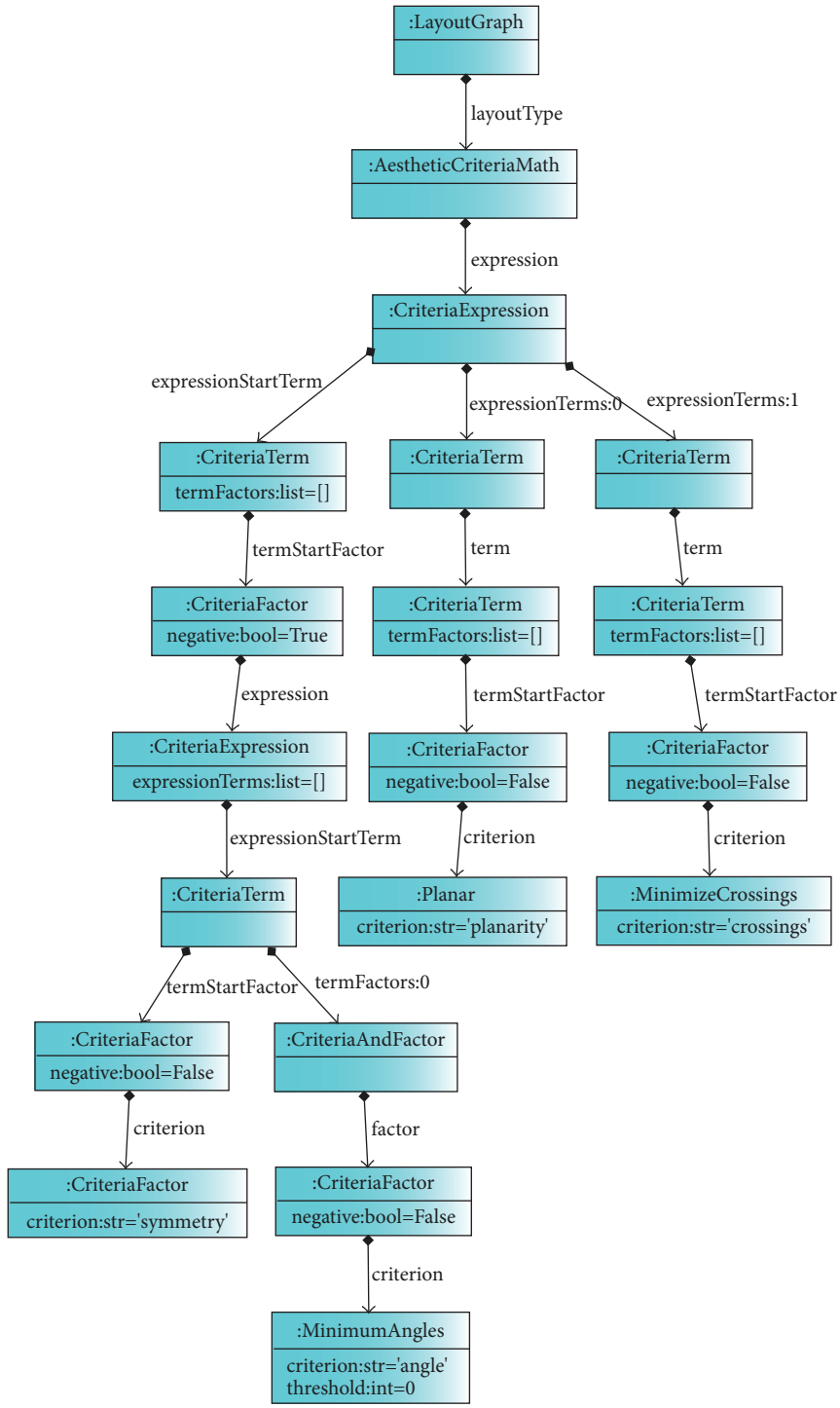


Figure 9: Model created by textX from the input shown in Listing 13.

algorithm or algorithms given a description conforming to it. This is why it heavily relies on the GRAD library, which contains implementations of the needed graph drawing and analysis algorithms.

Naturally, in terms of implementation, the easiest method is the one requiring the most specific input from the users, directly naming an algorithm and possibly configuring its parameters. In such a case, there is no need to analyze the graph or to take any additional steps. However, in order to achieve the desired goal using this method, a user would have to be fairly knowledgeable. Such users could also directly take advantage of one of the available implementations of the layout algorithms and completely skip the step of using the DSL. For them, the DSL is simply another alternative. Therefore, the remaining methods are much more interesting and potentially useful.

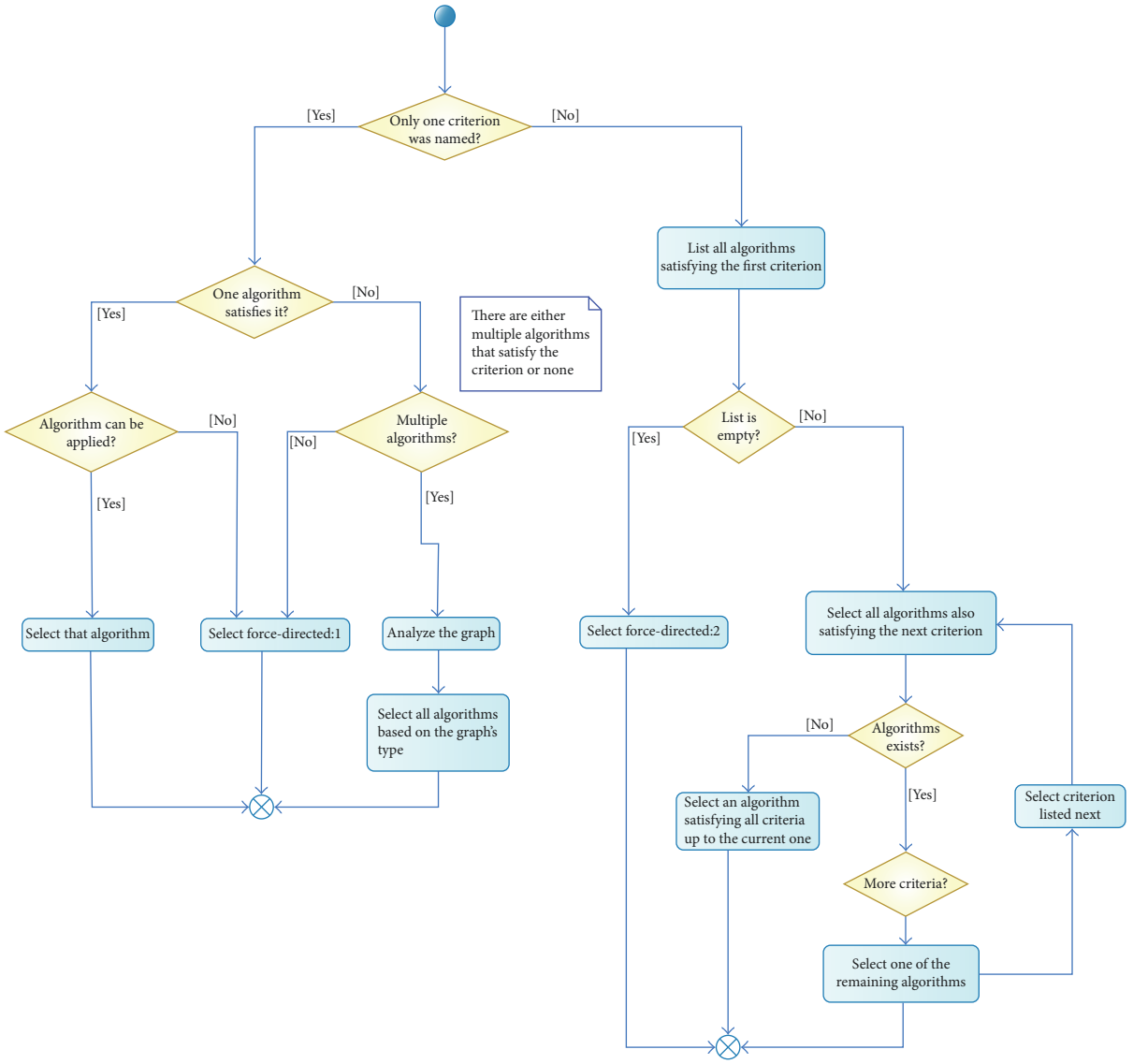


Figure 10: The process of selecting an algorithm based on a list of aesthetic criteria.

The first of them provides a way of defining how a graph should be laid out by naming one or more of the desirable properties of the drawing, that is, one or more aesthetic criteria, with the order in which they are listed defining their priorities. Like it was mentioned in the previous section, all graph layout algorithms (except for the most basic ones) were designed in accordance with at least one aesthetic criterion. Some, like the force-directed graph layout algorithms, tend to satisfy several criteria to some extent. However, if one specific criterion is particularly significant, these more general approaches are rarely the best choice. An algorithm focusing on that criterion will produce a drawing satisfying it to a greater extent. For example, while force-directed layout algorithms tend to produce crossing-free drawings of planar graphs, there is no guarantee. Planar straight-line drawing, on the other hand, will always do so. Similarly, force-directed algorithms tend to produce symmetric drawings, but they will almost certainly exhibit far less symmetries than those created by a symmetric algorithm.

Tables 4 and 5 show which criteria the available algorithms conform to. The symbol ± means that the algorithm satisfies the criterion to some extent. It can be noticed that not all force-directed algorithms were taken into consideration, since the organic one, which was designed in accordance with the greatest number of aesthetic criteria, was singled out.

Having everything mentioned in mind, a process of selecting an algorithm based on the list of aesthetic criteria was designed. In order to simplify its understanding, it was visualized using an activity diagram shown in Figure 10. The selection process consists of the following rules:

(i) If only one criterion was named, an algorithm specifically designed with it in mind is sought.

  (a) If there is such algorithm and it can be applied to the given graph, it is selected. For example, if

Table 4: Algorithms and the more general aesthetic criteria.

Algorithm	Planarity	Flow	Symmetry	Distribution of nodes
Organic	±	−	±	+
Radial tree	+	−	±	±
Level-based tree	+	+	±	−
Balloon tree	+	−	±	−
Hierarchical	±	+	−	−
Symmetrical	−	−	+	−
Tutte	+	−	−	−
Convex	+	−	−	−
Orthogonal	+	−	−	−
Circular	±	−	−	−

Table 5: Algorithms and aesthetic criteria related to edges.

Algorithm	Bends	Angle	Variation	Lengths
Organic	+	−	±	±
Radial tree	+	−	+	+
Level based tree	+	−	+	+
Ballon tree	+	−	−	−
Hierarchical	±	−	−	−
Symmetrical	+	−	−	−
Tutte	+	−	−	−
Convex	+	±	−	−
Orthogonal	±	+	−	−
Circular	+	−	−	−

the only criterion is symmetry, the symmetrical one is chosen.

(b) If only one algorithm addresses that criterion, but just to some extent, it is chosen if it can be applied on the graph.

(c) If several algorithms produce drawings which conform to that criterion, properties of the graph are also taken into account. For example, if the graph is a tree and planarity is the selected criterion, a tree drawing algorithm is picked.

(d) If none of the algorithms that satisfy the criterion can be applied, the organic force-directed algorithm is selected, unless the graph has over 1000 vertices. If that, however, is the case, the ISOM algorithm is chosen instead, due to its exceptional computation efficiency.

(ii) If more than one criterion is specified, an algorithm whose results conform to the greatest number of them is sought. The order in which the criteria were listed is an important factor of the selection process; it defines their priority. Naturally, the firstly mentioned criterion has the highest priority. So, the algorithm which is to be picked should satisfy the first criterion as much as possible, which narrows down the set of possible candidates. For example, if flow, planarity, and even distribution of nodes are listed in that order and graph is not a tree, a hierarchical algorithm will be chosen. If the order is then changed, with the criteria list being distribution of nodes, planarity, and flow, the organic algorithm will be picked. It should be noted that it is possible to specify a combination of criteria such that not all of them can be satisfied. Criteria with lower priorities are then ignored.

The second layout specification method relying on aesthetic criteria is the one allowing the usage of logical (and/or/not) operators. Its implementation is based on data provided in Tables 4 and 5. If, for example, the input corresponds to the last example seen in Listing 4, it is first checked if there is an algorithm whose drawings do not conform to neither planarity nor uniform flow criteria. As can be seen in Table 4, the symmetrical algorithm satisfies that condition and is then selected.

Finally, the last method enables the users to name a general style of the drawing, for example, hierarchical or circular, or to say that a graph should be laid out automatically. If a style was specified, the best available implementation of an algorithm belonging to the appropriate class is selected. This means that an algorithm generally producing nice drawings while also being reasonably efficient is chosen. If a graph should be laid out automatically, it is analyzed with the goal of detecting properties required by the available specialized algorithms (like planar straight-line or the tree ones). The existence of the more restrictive properties is determined first. So, if a graph is a tree (which means that it is also planar),

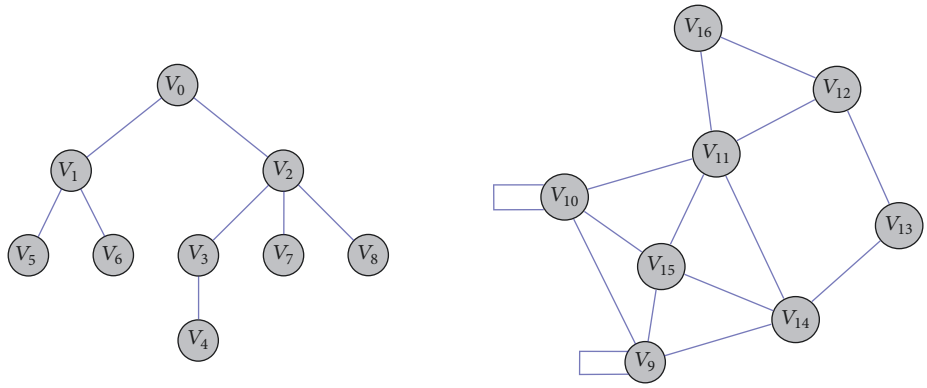


Figure 11: An example of an automatically laid out graph.

a tree drawing algorithm is to be chosen and not a more general planar one. If it represents a hierarchy and is not a tree, a hierarchical algorithm is selected. If the graph does not have any of the sought features, a force-directed algorithm is singled out, since members of this class can be applied to any graph. Additionally, if the given graph has over 1000 vertices, the set of possible picks is limited to more efficient implementations. An example of an automatically laid out graph, consisting of two disjoint subgraphs and containing loops is shown in Figure 11.

As it was already mentioned, GRAD strives to make integration with any graphical editor a very easy task. Making use of any of GRAD's layout methods can be by writing just a few lines of code. All that needs to be passed to GRAD in order to lay out a graph using the DSL are lists of its vertices and edges and a text that should conform to the rule's of the language. The result is then returned as a mapping from each vertex to its position and from each edge to a list of positions of its nodes. One example of GRAD's usage will be shown in the next section.

5. Example of Usage and Evaluation

The previous sections presented different aspects of the GRAD library, including the processes of automatically selecting, configuring, and applying a layout algorithm and its domain-specific language. This section will focus on showing a user study whose main goal was to verify that readability and understandability of diagrams automatically generated by a graphical editor using GRAD are on average better than of those generated by well-known commercial and open-source tools. The graphical editor which relies on GRAD and was used in the study is Kroki's lightweight UML editor.

5.1. Kroki Mockup Tool and GRAD. Kroki [73] is an open-source tool for interactive development of business applications. It supports two different notations: UML based and mockup based, implemented by its lightweight UML editor and mockup editor, respectively. Mockup editor allows the users to create forms by dragging and dropping user interface components (text fields, combo-boxes, etc.) from a palette. Every designed form is represented with a UML class in a UML class diagram editor and vice versa. Both editors can be used interchangeably to define a single application, which means that a specification created using the mockup tool might later be opened using the UML editor. In that case, it is necessary to automatically create, link, and lay out UML classes corresponding to the forms. Additionally, Kroki supports importing models created using general purpose tools, which can be quite large and contain hundreds of elements.

Figure 12 shows an imported model opened using Kroki's UML class diagram editor and automatically laid out using GRAD. The diagram is relatively large, containing 26 classes and 40 links.

5.2. The User Study. The evaluation of the GRAD library was carried out by conducting a user study where the participants (programmers with up to 25 years of experience) were asked to rate visual attractiveness and readability of the same models laid out and visualized using different tools, as well as understandability of several code samples for triggering the process of laying out a graph. The goal of the study was to confirm that

(1) layouts of diagrams opened in a graphical editor which relies on GRAD are on average more appealing than those of diagrams laid out and visualized with leading commercial and open-source modeling tools;

(2) GRAD's layout API is easier to use and understand compared to those of the most popular Java graph analysis and drawing libraries;

(3) a large percent of developers (close to 50%) would prefer to use GRAD's external DSL when invoking a layout algorithm to any of the other methods.

5.2.1. The Evaluation Procedure. The evaluation was based on visualizing a relatively complex UML model (containing over 200 classes belonging to 31 packages) using four different tools supporting UML class diagram modeling: SAP Sybase PowerDesigner, MagicDraw, Kroki (which uses GRAD), and Papyrus. PowerDesigner and MagicDraw are industry-leading software and system modeling tools, while Papyrus is an open-source modeling environment, with increasing

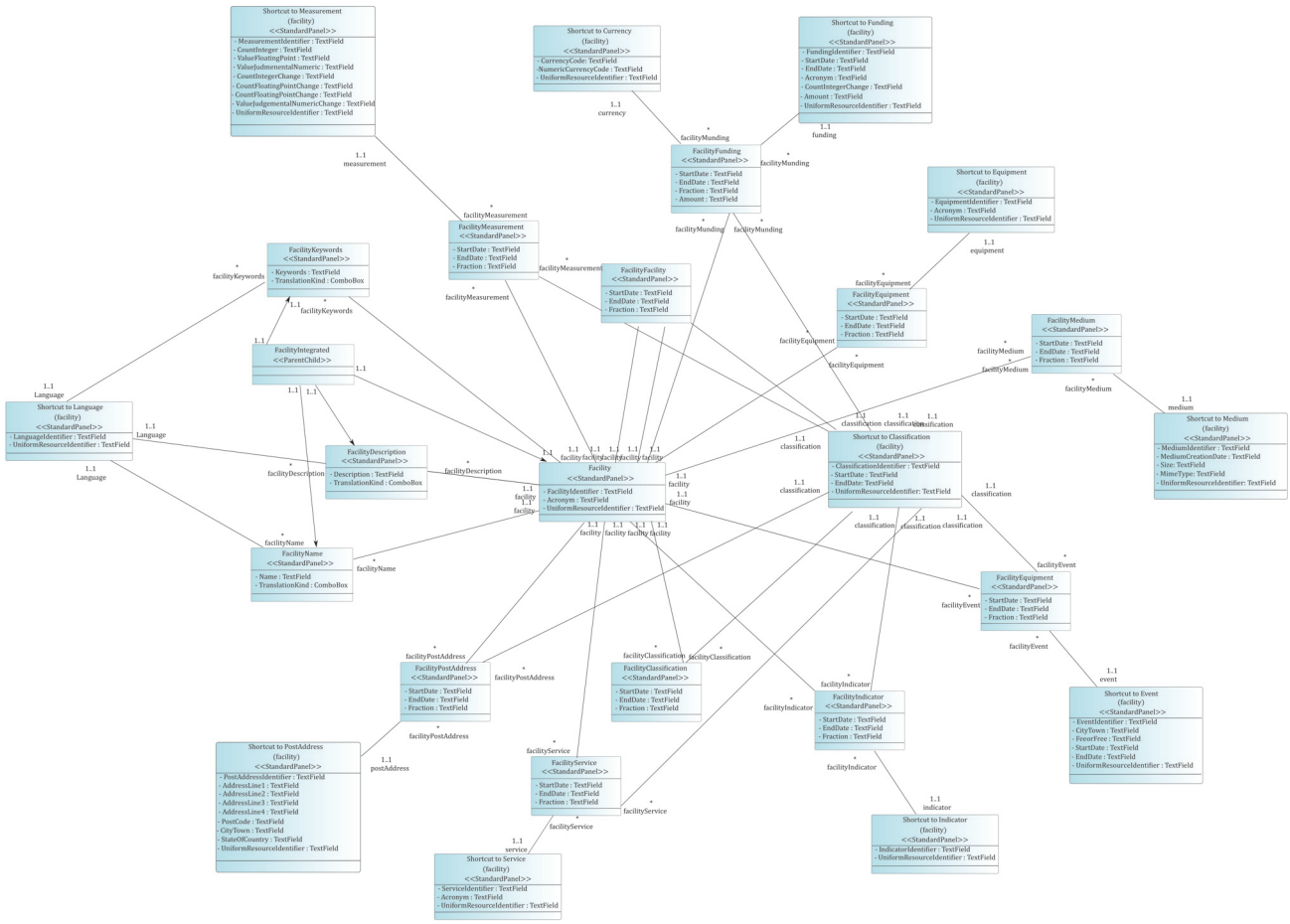


Figure 12: An imported model [15] opened in Kroki's UML diagram editor and automatically laid out using GRAD.

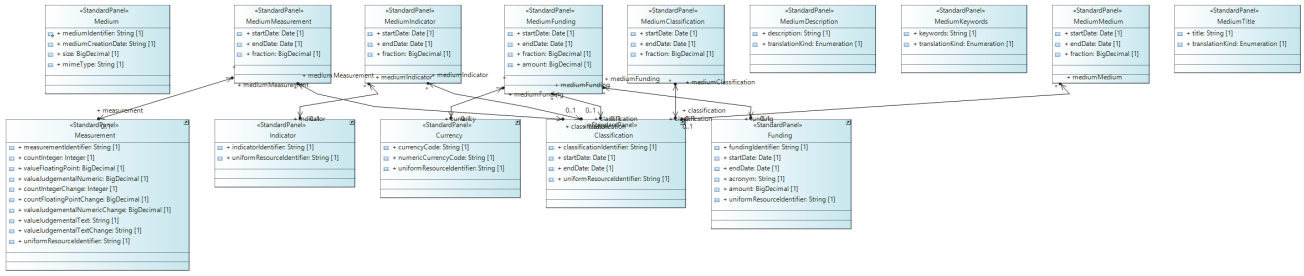
popularity. Ten of the 31 packages were selected and opened in each of the tools. PowerDesigner and Kroki automatically laid out the diagrams, while in case of MagicDraw and Papyrus the process had to be invoked manually. Both MagicDraw and Papyrus do, however, provide completely automated layout functionalities (without forcing the users to pick or configure an algorithm themselves). All properties of the diagrams' elements, except for their positions, were set to equal values for each of the tools, in order to avoid the impact of personal preferences of the participants on the results. Without being told which tool generated which diagram, the participants were shown ten sets of four diagrams (a diagram automatically laid out with each tool for every package) and were given two tasks:

(1) To grade their overall aesthetics using a 1–5 Likert scale, represented by a gradation between the choices of very poor and excellent.

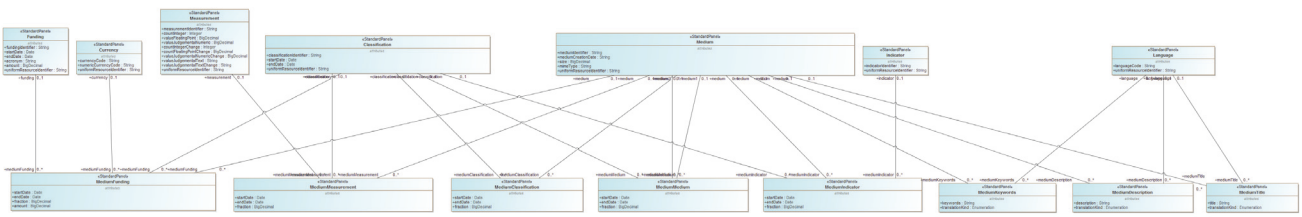
(2) To try to solve a relatively simple task and rate the difficulty of doing so. These tasks included finding paths between classes, counting the number of edges connecting two classes (thus testing how well the tools handle multiple edges), counting the number of edges involving a certain class and so on. The participants could pick an answer ranging from very difficult (1) to very easy (5).

The latter mentioned type of questions was introduced in order to compare the readability of the diagrams, since a diagram can be nice to look at, but impractical. For example, a diagram where multiple edges completely overlap could seem more aesthetically pleasing compared to one showing all such edges, while also hiding some of the important relationships between classes. While choosing the ten packages to use for the evaluation, attention was paid to selecting both smaller and simpler ones with just 5 or 6 classes and under 10 links and relatively complex ones, with over 20 classes and approximately twice as many links. As an example, Figure 13 presents one of the ten packages laid out using the four mentioned tools.

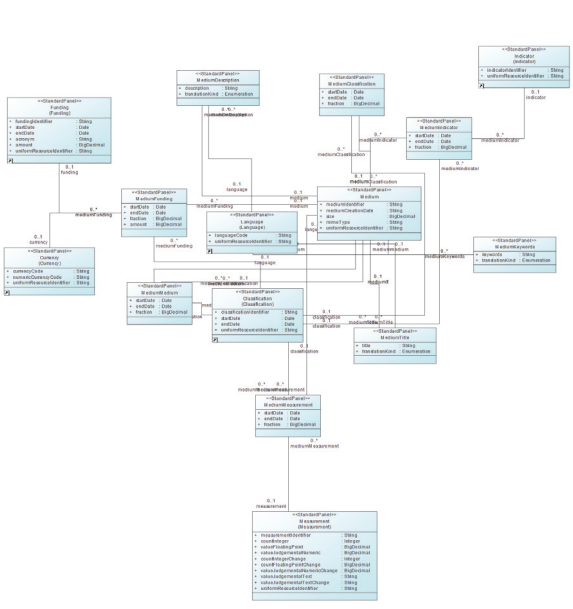
In the second phase of the study, the participants were shown five different code samples where a layout algorithm is invoked and results of its execution are retrieved. The samples included those using the mentioned JGraphX, JUNG, and prefuse libraries, while the remaining two used GRAD, one with and one without taking advantage of the DSL. All diagrams and code samples used in the study as well as more detailed results are available online [74].



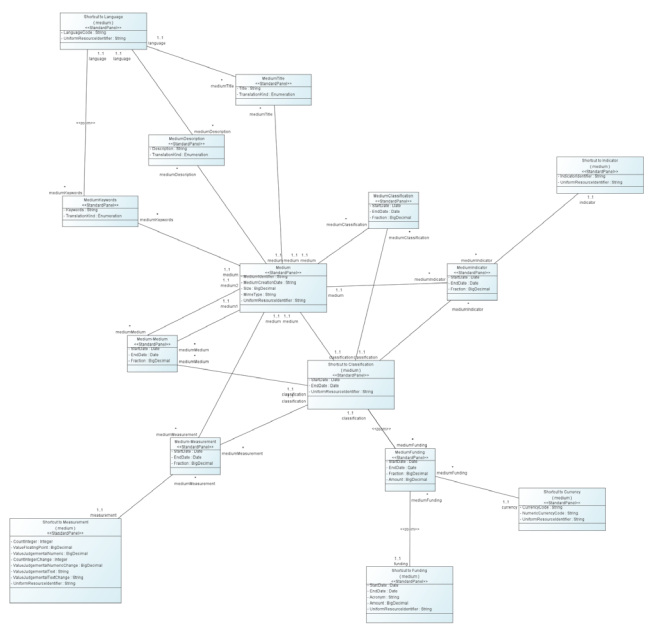
(a) Papyrus



(b) MagicDraw



(c) PowerDesigner



(d) Kroki and GRAD

Figure 13: The models' medium-sized package visualized using four different tools.

Table 6: Average values of provided answers to the background information questions.

Age	Years of programming experience in Java	Years of modelling experience
31.4	10	5.6

5.2.2. Participants. All participants, the total number of which was 31, were software engineers with some experience in UML modeling. At the start of the evaluation, they were asked to provide some general information about themselves and rate their skills in relevant fields on a scale of 1–10. The average results of these questions are shown in Tables 6 and 7.

The participants were asked if they would, if they were developing a graphical editor rather implement a layout algorithm themselves or look for existing solutions. A vast

Table 7: Results of the skill evaluation questionnaire.

Question	1	2	3	4	5	6	7	8	9	10	Average
How skilled you would say you are at modeling UML class diagrams?	3%	0%	0%	6%	0%	10%	16%	23%	29%	13%	7.71/10
Are you familiar with the area of graph drawing?	6%	10%	13%	6%	13%	16%	19%	13%	3%	0%	5.19/10
How would you rate your graphic design skills?	6%	6%	16%	16%	19%	10%	3%	10%	13%	0%	5.03/10

Table 8: Average aesthetics scores and standard deviations.

Model number	GRAD		PowerDesigner		MagicDraw		Papyrus	
	Average	SD	Average	SD	Average	SD	Average	SD
(1)	3.65	4.17	2.71	5.11	3.61	3.19	1.97	5.49
(2)	4.55	7.39	3.81	4.71	4.03	4.75	2.48	4.26
(3)	4.29	5.88	2.45	5.42	3.45	4.17	1.65	6.14
(4)	4.13	5.15	2.1	5.53	3.32	3.87	1.68	6.14
(5)	4.1	5.19	2.03	5.08	3.58	6.34	1.90	6.65
(6)	4.1	4.92	2.0	5.67	3.84	4.45	1.81	5.98
(7)	4.06	6.52	2.32	4.92	2.35	5.95	1.81	5.74
(8)	4.42	6.85	2.45	4.66	3.87	6.21	2.26	4.83
(9)	4.29	5.84	2.77	5.0	3.61	5.78	2.52	5.74
(10)	4.06	4.96	2.48	5.78	3.39	5.88	1.84	5.38

Table 9: Average task solving scores and standard deviations.

Model number	GRAD		PowerDesigner		MagicDraw		Papyrus	
	Average	SD	Average	SD	Average	SD	Average	SD
(1)	4.29	6.4	2.9	2.32	2.35	4.49	3.35	4.17
(2)	4.68	8.45	4.32	6.11	4.1	4.87	2.71	2.23
(3)	4.52	8.06	2.19	4.96	3.9	4.31	1.61	7.11
(4)	3.74	4.75	1.97	4.92	3.19	3.76	1.52	7.65
(5)	4.39	6.68	1.94	5.04	4.1	5.64	1.97	4.83
(6)	4.35	6.24	1.97	4.83	4.03	4.66	2.0	4.45
(7)	4.65	8.13	2.26	4.02	3.19	4.17	2.42	3.76
(8)	4.68	9.06	1.84	5.27	3.32	2.48	2.71	4.17
(9)	4.13	5.81	2.23	4.02	3.45	4.92	2.48	3.37
(10)	4.23	5.71	2.23	5.04	3.52	5.56	1.65	6.14

majority (84%) said that they would prefer to use existing implementations.

5.2.3. Results. Results of the first phase of the evaluation are shown in Tables 8 and 9. Average aesthetics scores with standard deviations can be seen in Table 8, while Table 9 contains assessments of difficulties of completing the presented tasks.

The average scores show that the participants found diagrams laid out using GRAD to be the most aesthetically pleasing. In fact, 9 out of 10 these diagrams received an average score of 4/5 or better. It can also be noted that the difference between GRAD and other tools is particularly evident in cases of larger diagrams, such as models 3, 4, 9, and 10. GRAD also received the highest scores with regard to how easy it was to solve the presented tasks.

The results of the evaluation of the ease of use of several graph drawing libraries are shown in Table 10. The participants were also asked to pick a code sample they would prefer to use, and the percentages of those selections are shown in Table 11.

It can be noticed that GRAD's regular method of calling a layout algorithm and retrieving the results, which does not use the DSL, received the highest average score. However, GRAD's second method, where the DSL is used, was selected by 42% of participants as the method they would prefer to use. Doing so requires learning the syntax of the DSL, which explains the fact that some participants did not give it a very high grade. On the other hand, it is the shortest of all methods, which almost half of the participants appreciated.

6. Conclusion

This paper presented a domain-specific language designed for simplifying the process of laying out a graph, as well as GRAD, our library for graph analysis, and drawing containing the algorithms necessary for the DSL's implementation. To the best of our knowledge, there is no other DSL focusing

Table 10: Average scores and standard deviations of code intuitiveness evaluation.

JGraphX		JUNG		GRAD		prefuse		GRAD DSL	
Avg	SD	Avg	SD	Avg	SD	Avg	SD	Avg	SD
3.06	3.82	3.48	5.78	4.06	6.27	2.45	3.31	3.94	4.31

Table 11: Percentage of users who would prefer to use the code sample.

JGraphX	JUNG	GRAD	prefuse	GRAD DSL
6%	16%	32%	3%	42%

exclusively on graph layout and some of the algorithms implemented in GRAD have not been implemented Java before.

GRAD's domain-specific language primarily aims to help users without extensive knowledge of graph drawing theory and algorithms lay out a certain diagram in accordance with their wishes. This is accomplished through naming a general style of the drawing or listing its desirable properties (or aesthetic criteria of the drawing). Based on such specification and properties of the graph, the most appropriate layout algorithm is automatically chosen, configured, and executed. The DSL can also be used by expert users to quickly select and configure an algorithm. Furthermore, if so is desired, subgraphs can be singled out and their layouts can be defined separately.

GRAD is currently being used in our Kroki mockup tool for laying out models which sometimes contain over 600 classes. A conducted user study showed that the participants generally rated diagrams whose elements were positioned using GRAD better than diagrams laid out using commercial tools (PowerDeigner and MagicDraw) and one of the most widely used open-source solutions, Papyrus. Furthermore, the study revealed that a vast majority of participants found GRAD's code samples more intuitive compared to the alternatives, JGraphX, JUNG framework, and prefuse.

As new algorithms are added to the GRAD library, the DSL will be expanded to support them. The automatic choice of the most appropriate algorithm will be enhanced, so that it is also checked if the newly implemented algorithm is the best option in a particular situation. Algorithms whose addition is planned include one or more orthogonal, polyline, and more sophisticated symmetric ones.

Additionally, integration with the popular Sirius tool for creation of graphical editors is also among planned future enhancements.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Computational Geometry. Theory and Applications*, vol. 4, no. 5, pp. 235–282, 1994.

[2] H. C. Purchase, D. Carrington, and J.-A. Allder, "Empirical evaluation of aesthetics-based graph layout," *Empirical Software Engineering*, vol. 7, no. 3, pp. 233–255, 2002.

[3] H. C. Purchase, "Effective information visualization: A study of graph drawing aesthetics and algorithms," *Interacting with Computers*, vol. 13, no. 2, pp. 147–162, 2000.

[4] R. Tamassia, G. D. Battista, and C. Batini, "Automatic Graph Drawing and Readability of Diagrams," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 18, no. 1, pp. 61–79, 1988.

[5] C. Bennett, J. Ryall, L. Spalteholz, and A. Gooch, "The aesthetics of graph visualization," in *Proceedings of the in Proceedings of the Third Eurographics conference on Computational Aesthetics in Graphics, Visualization and Imaging*, pp. 57–64, 2007.

[6] M. Petre, "Cognitive dimensions 'beyond the notation'," *Journal of Visual Languages and Computing*, vol. 17, no. 4, pp. 292–301, 2006.

[7] M. Schrepfer, J. Wolf, J. Mendling, and H. A. Reijers, "The impact of secondary notation on process model understanding," *Lecture Notes in Business Information Processing*, vol. 39, pp. 161–175, 2009.

[8] H. Purchase, J.-A. Allder, and D. Carrington, "User preference of graph layout aesthetics: A uml study," in *in Proceedings of the 8th International Symposium on Graph Drawing*, pp. 5–18, 2000.

[9] Provide a custom arrange-all, https://www.eclipse.org/sirius/doc/developer/extensions-provide_custom-arrange-all.html.

[10] M. Narkhede1, S. Patil, and V. Inamdar, "Comparative study of various graph layout algorithms," *International Journal of Emerging Trends and Technology in Computer Science*, vol. 3, 2014.

[11] L. Vismara, *Handbook of Graph Drawing and Visualization*, ch. 6, Chapman and Hall/CRC, 2007.

[12] P. Eades and P. Mutzel, *Handbook of Graph Drawing and Visualization*, ch. 3, Chapman and Hall/CRC, 2007.

[13] T. Nishizeki and S. Rahman, *Handbook of Graph Drawing and Visualization*, ch. 10, Chapman and Hall/CRC, 2007.

[14] N. Chiba, K. Onoguchi, and T. Nishizeki, "Linear algorithms for convex drawings of planar graphs," in *Progress in Graph Theory*, vol. 173, pp. 153–173, Academic Press, Toronto, ON, 1984.

[15] Cerif, http://www.eurocris.org/cerif/introduction/.

[16] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[17] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[18] H. Purchase, "Which aesthetic has the greatest effect on human understanding?" in *Graph Drawing Software*, vol. 1353 of *Lecture Notes in Computer Science*, pp. 248–261, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[19] M. K. Coleman and D. Stott Parker, "Aesthetics-based graph layout for human consumption," *Software: Practice and Experience*, vol. 26, no. 12, pp. 1415–1438, 1996.

[20] H. C. Purchase, J.-A. Allder, and D. Carrington, "Graph layout aesthetics in UML diagrams: user preferences," *Journal of Graph Algorithms and Applications*, vol. 6, no. 3, pp. 255–279, 2002.

[21] W. Huang, S. hee Hong, and P. Eades, "Layout effects: Comparison of sociogram drawing conventions," Tech. Rep., University of Sydney, 2005.

[22] R. Vaderna, I. Dejanovic, and G. Milosavljevic, "GRAD: A New Graph Drawing and Analysis Library," in *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016*, pp. 1597–1602, Poland, September 2016.

[23] R. Vaderna, G. Milosavljevic', and I. Dejanovic', "Laying out graphs using graph analysis and drawing library - grad," in *in Proceedings of the 25h International Computer Science Conference ERK 2016*, vol. B, pp. 51–54, 2016.

[24] R. Vaderna, G. Milosavljvic', and I. Dejanovic', "Graph layout algorithms and libraries: Overview and improvements," in *Proceedings of the in ICIST 2015 5th International Conference on Information Society and Technology Proceedings*, pp. 55–60, 2015.

[25] M. Patrignani, *Handbook of Graph Drawing and Visualization*, ch. 1, Chapman and Hall/CRC, 2007.

[26] S. Skiena, *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, ch. 5, Addison-Wesley Publishing Company, 1990.

[27] Graph crossing number, http://mathworld.wolfram.com/GraphCrossingNumber.html.

[28] H. Purchase, *Evaluating Graph Drawing Aesthetics: defining and exploring a new empirical research area*, ch. 8, Idea Group Publishing, 2004.

[29] The dot language, http://www.graphviz.org/doc/info/lang.html.

[30] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[31] A. Bergel, S. Maass, S. Ducasse, and T. Girba, "A domain-specific language for visualizing software dependencies as a graph," in *Proceedings of the 2nd IEEE International Working Conference on Software Visualization, VISSOFT 2014*, pp. 45–49, Canada, September 2014.

[32] Dsl based approach to input graph data in graph theory based java programs, https://sanaulla.info/2013/07/19/dsl-based-approach-to-input-graph-data-in-graph-theory-based-java-programs.

[33] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for easy and efficient graph analysis," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012*, pp. 349–362, UK, March 2012.

[34] R. Tamassia, Ed., *Handbook of graph drawing and visualization*, CRC Press, Boca Raton, FL, 2014.

[35] C. Buchheim, M. Chimani, C. Gutwenger et al., *Handbook of Graph Drawing and Visualization*, ch. 2, Chapman and Hall/CRC, 2007.

[36] C. Gutwenger and P. Mutzel, "Planar polyline drawings with good angular resolution," in *Proceedings of the Graph Drawing Symposium 1998*, pp. 167–182, Springer.

[37] H. C. Purchase, "Metrics for Graph Drawing Aesthetics," *Journal of Visual Languages & Computing*, vol. 13, no. 5, pp. 501–516, 2002.

[38] R. Tamassia, "On embedding a graph in the grid with the minimum number of bends," *SIAM Journal on Computing*, vol. 16, no. 3, pp. 421–444, 1987.

[39] A. Papakostas and I. G. Tollis, "Efficient orthogonal drawings of high degree graphs," *Algorithmica. An International Journal in Computer Science*, vol. 26, no. 1, pp. 100–125, 2000.

[40] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989.

[41] Graph automorphism, http://mathworld.wolfram.com/GraphAutomorphism.html.

[42] A. Rusu, *Handbook of Graph Drawing and Visualization*, ch. 5, Chapman and Hall/CRC, 2007.

[43] P. Healy and N. Nikolov, *Handbook of Graph Drawing and Visualization*, ch. 13, Chapman and Hall/CRC, 2007.

[44] J. Six and I. Tollis, *Handbook of Graph Drawing and Visualization*, ch. 9, Chapman and Hall/CRC, 2007.

[45] S. Kobourov, *Handbook of Graph Drawing and Visualization*, Chapman and Hall/CRC, 2013.

[46] W. T. Tutte, "How to Draw a Graph," *Proceedings of the London Mathematical Society*, vol. 3-13, no. 1, pp. 743–767, 1963.

[47] P. Eades, "A heuristic for graph drawing," *Congressus Numerantium*, vol. 42, pp. 149–160, 1984.

[48] R. Davidson and D. Harel, "Drawing Graphs Nicely Using Simulated Annealing," *ACM Transactions on Graphics*, pp. 301–331.

[49] B. Meyer, "Self-Organizing Graphs — A Neural Network Perspective of Graph Layout," in *Graph Drawing Software*, vol. 1547 of *Lecture Notes in Computer Science*, pp. 246–262, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[50] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[51] H. Carr and W. Kocay, "An algorithm for drawing a graph symmetrically," *Bulletin of the Institute of Combinatorics and Its Applications*, vol. 27, pp. 19–25, 1999.

[52] N. Chiba, T. Yamanouchi, and T. Nishizeki, *Progress in Graph Theory*, ch. 5, Academic Press, 1984.

[53] J. M. Six and I. G. Tollis, "A framework and algorithms for circular drawings of graphs," *Journal of Discrete Algorithms*, vol. 4, no. 1, pp. 25–50, 2006.

[54] R. Tamassia and I. G. Tollis, "A unified approach to visibility representations of planar graphs," *Discrete & Computational Geometry*, vol. 1, no. 1, pp. 321–341, 1986.

[55] Graph analysis and drawing library, https://www.gradlibrary.net.

[56] Y. Hu, "Efficient and high quality force-directed graph drawing," *The Mathematica Journal*, vol. 10, pp. 37–71, 2005.

[57] H. de Fraysseix and P. Ossona de Mendez, "Trémaux trees and planarity," *European Journal of Combinatorics*, vol. 33, no. 3, pp. 279–293, 2012.

[58] J. M. Boyer and W. J. Myrvold, "On the cutting edge: Simplified o(n) planarity by edge addition," *Journal of Graph Algorithms and Applications*, vol. 8, no. 3, pp. 241–273, 2004.

[59] K. S. Booth and G. S. Lueker, "Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms," *Journal of Computer and System Sciences*, vol. 13, no. 3, pp. 335–379, 1976.

[60] J. E. Hopcroft and R. E. Tarjan, "Dividing a graph into triconnected components," *SIAM Journal on Computing*, vol. 2, no. 3, pp. 135–158, 1973.

[61] S. Fialko and P. Mutzel, "A new approximation algorithm for the planar augmentation problem," in *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 260–269, Society for Industrial and Applied Mathematics, 1998.

[62] B. D. McKay, "Practical graph isomorphism," *Proceedings of the Tenth MANitoba Conference on Numerical MAThematics and COMputing, Vol. I (Winnipeg, MAN., 1980)*, vol. 30, pp. 45–87, 1981.

[63] M. Mernik, D. Hrncic, B. R. Bryant, and F. Javed, "Applications of grammatical inference in software engineering: domain specific language development," *Scientific applications of language methods*, vol. 2, pp. 421–457, 2011.

[64] I. Čeh, M. Črepinšek, T. Kosar, and M. Mernik, "Ontology driven development of domain-specific languages," *Computer Science and Information Systems*, vol. 8, no. 2, pp. 317–342, 2011.

[65] T. Kosar, M. Mernik, and J. C. Carver, "Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments," *Empirical Software Engineering*, vol. 17, no. 3, pp. 276–304, 2012.

[66] M. Fowler, *Domain Specific Languages*, Addison-Wesley Professional, 2010.

[67] W. Cazzola and E. Vacchi, "Language components for modular DSLs using traits," *Computer Languages, Systems and Structures*, vol. 45, pp. 16–34, 2016.

[68] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković, "TextX: A Python tool for Domain-Specific Languages implementation," *Knowledge-Based Systems*, vol. 115, pp. 1–4, 2017.

[69] T. Kosar, S. Bohra, and M. Mernik, "Domain-Specific Languages: A Systematic Mapping Study," *Information and Software Technology*, vol. 71, pp. 77–91, 2016.

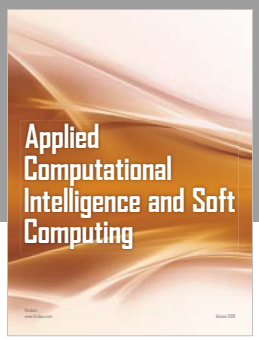
[70] M. Nosáľ, J. Porubän, and M. Sulír, "Customizing host IDE for non-programming users of pure embedded DSLs: A case study," *Computer Languages, Systems and Structures*, vol. 49, pp. 101–118, 2017.

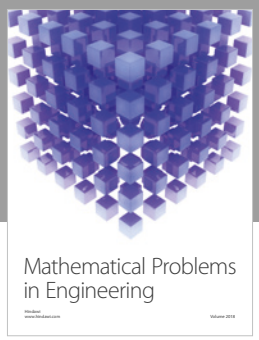
[71] A. Barišić, V. Amaral, and M. Goulão, "Usability driven DSL development with USE-ME," *Computer Languages, Systems and Structures*, 2017.

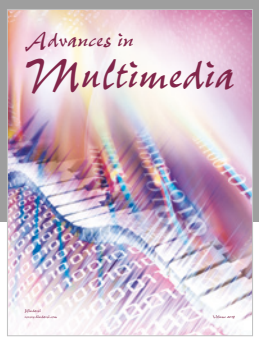
[72] Graph layout dsl, https://github.com/renatav/GraphDrawing/blob/master/GraphLayoutDSL/src/language/layout.tx.

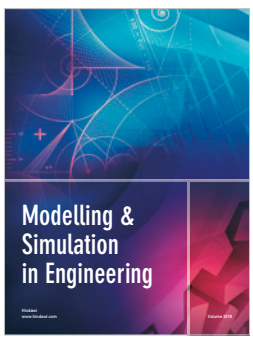
[73] G. Milosavljevic, M. Filipovic, V. Marsenic, D. Pejakovic, and I. Dejanovic, "Kroki: A mockup-based tool for participatory development of business applications," in *Proceedings of the 12th IEEE International Conference on Intelligent Software Methodologies, Tools and Techniques, SoMeT 2013*, pp. 235–242, Hungary, September 2013.

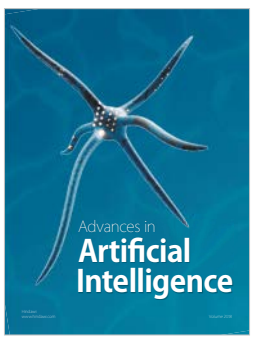
[74] R. Vaderna, *Grad Evaluation Results, Mendeley Data*, 2017.

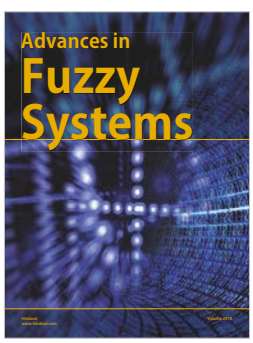


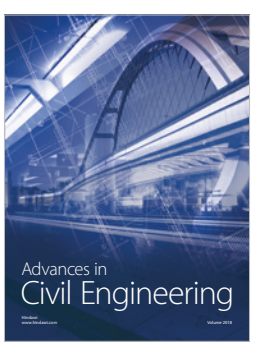


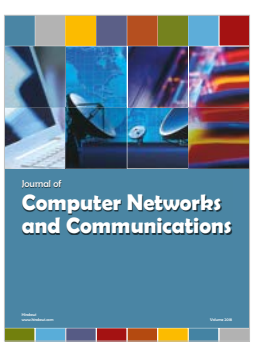


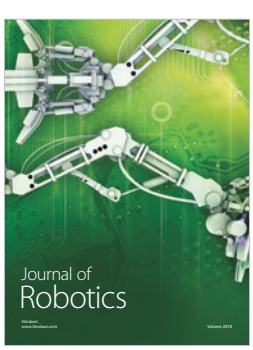


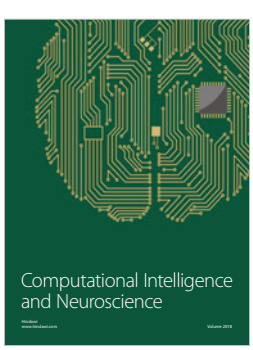












Hindawi

Submit your manuscripts at
www.hindawi.com