Christian-Albrechts-Universität zu Kiel

Student Research Project

KIML-EVOL

Evolutionary Meta Layout for KIELER

Björn Duderstadt

June 1, 2011

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl. Inf. Miro Spönemann

Abstract

The scope of this student research project is to find an approach to meta layout of graphical models, using evolutionary algorithms to learn from user feedback on created layouts.

This approach strives for two goals. Firstly, it may aid users who want a personalized automatic layout of graphical models. So far, automatic layout primarily relied on *a priori* guesses about the users' personal preferences with respect to layout. If the users wanted alternative automatic layout, the only way was by manually specifying layout options. The approach presented here iteratively evolves layout configurations which lead to different layout proposals. The users are asked for feedback on the proposals. Configurations that lead to more "appealing" layout results are promoted. This method gives users the possibility to administer automatic layout of graphical models without needing to understand the interaction of layout options and their effect on layout algorithms.

Secondly, this approach gives developers of layout algorithms a means to evaluate them. It can help to find appropriate default settings for newly developed algorithms.

This thesis presents a first implementation of this evolutionary approach to meta layout which interacts with other tools from the KIELER project.

Contents

List of Figures

Listings

Abbreviations

ANN	Artificial Neural Network
CAU	Christian-Albrechts-Universität zu Kiel
EA	evolutionary algorithm
EMOO	Evolutionary Multiobjective Optimization
EP	Evolutionary Programming
ES	Evolution Strategies
EVOL	Evolutionary Variation and Optimizing of Layout options
GA	Genetic Algorithm
GP	Genetic Programming
GrAna	KIELER Graph Analysis
GUI	graphical user interface
IC	integrated circuit
ID	identifier
IDE	integrated development environment
IEC	Interactive Evolutionary Computing
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KIEM	KIELER Execution Manager
KIML	KIELER Infrastructure for Meta Layout
LISP	List Processing
MDSD	Model-Driven Software Development
MOEA	Multiobjective Optimization using Evolutionary Algorithms
MOO	multiobjective optimization

MVC Model-View-Controller

TSP Travelling Salesman Problem

UI user interface

UML Unified Modeling Language

WYSIWYG What You See Is What You Get

1 Introduction

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is a research project about enhancing the graphical model-based design of complex systems. KIELER is developed by the Real-Time and Embedded Systems Group of the University of Kiel.

One of the essential subprojects of KIELER is the KIELER Infrastructure for Meta Layout (KIML), which deals with the automatic layout of graphical models. Automatic layout is intended to exonerate users of graphical modeling tools following the WYSIWYG principle from the cumbersome task of manual layout. While KIELER supports different types of diagrams that need different types of layout, KIML serves as the link that connects graphical diagram editors and layout algorithms.

The automatic layout logic is based on *a priori* knowledge about user preferences for layout of the graphical model at hand, as well as on user-defined options. There are many options that can be set manually in order to configure the automatic layout process in detail. Without advanced knowledge this can be a cumbersome and frustrating task. Moreover, since automatic layout is intended to disburden the user, the need to adjust options in detail is contrary to its goal. The better the default settings reflect the users' preferences, the more useful is KIML.

The capability of learning layout preferences from user feedback could enable users to obtain pleasing layout without having to be concerned about different layout algorithms or layout options, therefore it may be a useful improvement of meta layout.

The goal of this work is to develop an approach to learn from user feedback on created layouts in KIELER. Evolutionary algorithms (EAs) shall be used to optimize layout options that control the creation of layouts.

This thesis presents an add-on to KIML, namely *KIML-EVOL*, where EVOL stands for Evolutionary Variation and Optimizing of Layout options. KIML-EVOL is an approach that uses interactive evolutionary algorithms to learn user preferences for layout options, so that they can later be used by KIML as default settings. This is done by proposing layout examples to the user. The judgment supplied by the user influences the generation of further layout proposals. However, it is not the layout proposals themselves that are evolved, but the parameter values for the layout algorithms and the choice of layout algorithms.

Implemented as an Eclipse plug-in, KIML-EVOL can be added to the KIELER framework to interact with KIML.

The rest of this paper is organized as follows: Chapter 1 introduces the concepts and terminology of EAs, aesthetics metrics, and meta layout. Furthermore, it introduces the technologies used in this project, and presents work related to this thesis.

A solution is proposed in Chapter 2, and finally, in Chapter 5, conclusions are drawn and ideas for future work are presented.

1.1 Evolutionary Algorithms

This section introduces the area of evolutionary algorithms (EAs). It presents the major types of EAs and the related terminology.

EAs are a family of standardized, stochastic algorithms that aim to find solutions to optimization problems. They do so by imitating some of the mechanisms that biologists believe to be effective in the evolution of species. Taking its inspiration from biology, the terminology of EAs borrows many biological terms. While the use of EAs also might help to build a deeper understanding of how the biological process of evolution works, that would only be a side-effect. The main goal is not to simulate natural evolution as accurately as possible, but rather only as much as is necessary to solve the optimization problems—which from a biologist view might be considered a severe oversimplification.

EAs are especially applicable to problems into which one does not yet have much insight. If there is more reliable knowledge about a problem, more sophisticated algorithms could be tailored instead. By purposefully exploiting the available knowledge, these would usually find better solutions in shorter time than an EA would. The advantage of EAs is that they often find sufficiently good solutions in reasonable time. However, it can usually not be guaranteed that they find optimal solutions.

Historically, there have been several independent approaches that have led to different "flavors" or types of EAs, *cf.* Whitley [36, p. 817, p. 823]. Beginning in the 1950's, first applications appeared. Later, attempts have been made to emphasize the similarities among the major types and to standardize the terms, see for example Spears *et al.* [30].

The overall process of EAs basically consists of a loop which is repeated many times. In each iteration, candidate solutions to the specified optimization problem—so-called *individuals*—are generated and evaluated. The constitution of such an individual is subject to basic design decisions. It depends on the problem at hand and also on the type of EA that is applied. A useful concept is to distinguish between the *genotype*, *i.e.* the genetic makeup of an individual, and its *phenotype*, *i.e.* the appearance it shows when it is evaluated as a candidate solution.

Specific operations are used that serve to pass advantageous features on to the individuals of the next iteration, while other operations aim on extinguishing disadvantageous features. This is inspired by natural selection, where fitter individuals have a higher chance of surviving long enough in their environment to pass their genetic makeup successfully onto their progeny, while less fit individuals tend to become eliminated by their harsh environment. In EA, the role of the environment in which the individuals struggle for existence is played by the problem that is to be solved.

While the above-mentioned overall process is more or less the same in all EAs, the

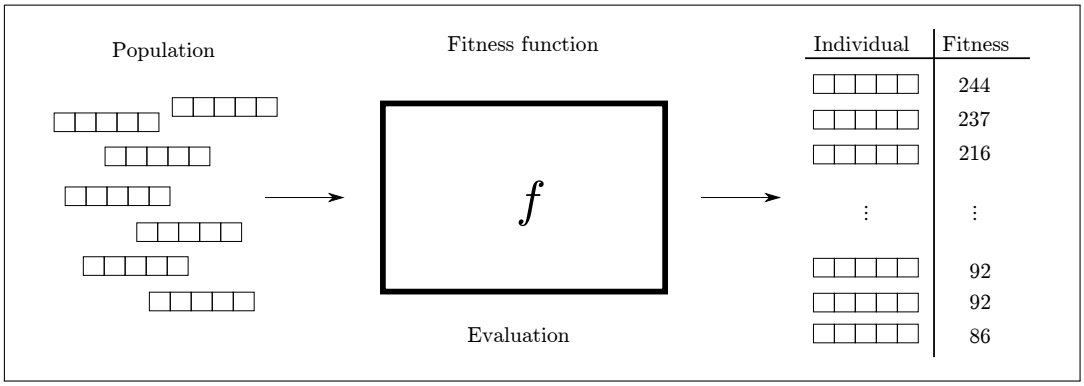


Figure 1.1: Fitness evaluation in EAs. The fitness function f is used to assign a fitness value to each member of the population.

implementation of each phase of the process can vary heavily from one algorithm to another.

The main phases typically used in EAs are initialization, evaluation, selection, reproduction, survival, and termination.

Initialization An initial population of n individuals is created. Typically, these are random individuals, *i. e.* they consist of random elements.

Evaluation The *fitness* (or quality) of each individual in the population is determined. This is typically expressed by some positive real number or integer, where a higher value indicates a fitter individual. Figure 1.1 illustrates the evaluation phase. The function f that maps individuals to their corresponding fitness values is called the *fitness function*. It is often metaphorically referred to as the *fitness landscape*, where the "mountain peaks" represent (local) maxima of the fitness function. Individuals that are located on these peaks are therefore considered "good" solutions. On the contrary, individuals down in the valleys are "poor" solutions. The appearance of the landscape depends on the problem; it can be arbitrarily jagged or smooth. Also the number of valleys, hills and peaks can vary heavily among different problems.

Survival Which individuals shall survive the current iteration is decided in the survival phase. The individuals with lowest fitness die, the ones with high fitness survive. They form the next generation. Some implementations also keep the fittest individuals of the preceding generation.

Selection Some $k \leq n$ individuals are chosen as candidates for reproduction. Typically, individuals with higher fitness are given a greater chance to be selected.

Reproduction From the k selected individuals, offspring are created by using one or both of the following reproductive operations: *crossover* and *mutation*. While

crossover serves to produce new combinations of existing variants, mutation randomly introduces new variants.

Termination The evolutionary process is repeated until some specified *stop criterion* is met. A simple method is to stop after a fixed number of generations has been calculated. Alternatively, the process could be stopped if the relative increase of the average fitness falls below a certain value for a specified number of iterations. In other words, the process is stopped after there has not been significant improvement for some generations.

As mentioned above, there are many different ways to implement these phases. For each phase, there have been studies which investigate how to do it better, *e. g.* Blickle and Thiele [4] compare different selection schemes. However, there is no "one size fits all" EA. Instead, it depends on the problem in question, which type of EA serves best, and how parameters such as mutation rate and population size need to be set. Although there are some rules of thumbs, the performance of an EA used on a specific problem cannot always be predicted easily and may need some twiddling with parameters.

1.1.1 Types of evolutionary algorithms

In the following paragraphs, the major types of EAs are introduced shortly, namely Genetic Algorithms (GAs), Evolution Strategies (ES), Evolutionary Programming (EP), and Genetic Programming (GP). Besides these types, hybrid forms and mavericks exist.

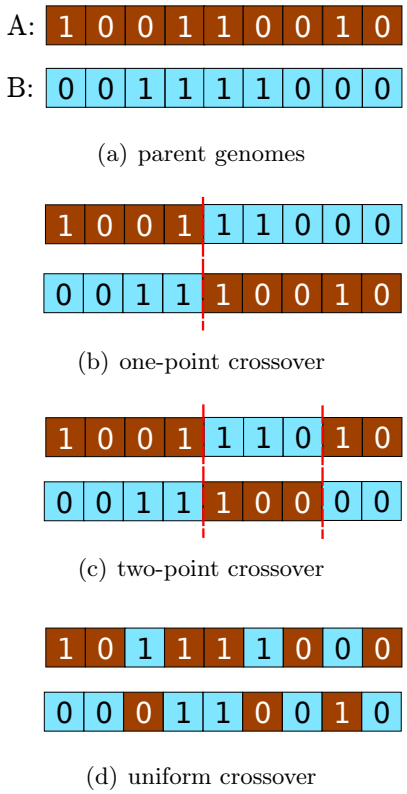
For a general overview of EAs, see Spears *et al.* [30]. Another overview, which focuses on practical issues, can be found in Whitley [36]. For a more detailed introduction to both GAs and ES consult Dianati *et al.* [9].

Genetic Algorithms

In GAs, we find a clear distinction between genotype and phenotype. Candidate solutions to a given optimization problem are encoded as bit vectors. A bit vector is called a *genome* or *individual*, and each bit is called a *gene*.

A vector of individuals is called a *population*.

The most important operation used in GAs is *crossover*. By crossover, two offspring genomes are created from two given genomes. This is done by copying the genomes and then splitting the copies at one or more random positions, the split positions. The sequences are swapped at the split positions. Besides crossover variants with a fixed number of split positions, there is also a variant where each position independently has the same chance to be a split position. In other words, for each pair of genes, it is decided whether to swap them or not. This is called uniform crossover. Three typical crossover variants for GAs are illustrated in Figure 1.2, namely one-point crossover or single-point crossover, two-point crossover, and uniform crossover. According to Dianati *et al.* [9], single-point crossover is the most applied crossover variant.



(a) parent genomes

(b) one-point crossover

(c) two-point crossover

(d) uniform crossover

Figure 1.2: Example illustrating crossover in a GA: (a) two parent genomes A and B, (b) offspring created by one-point crossover, (c) offspring created by two-point crossover, and (d) offspring created by uniform crossover.

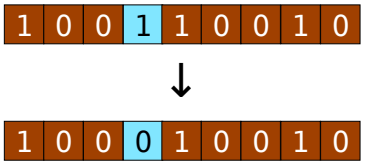


Figure 1.3: Example illustrating mutation in a GA. In this example, the displayed genome experiences a mutation of one of its genes (highlighted). Its value changes from 1 to 0.

Listing 1.1: Typical genetical algorithm

```
initialize();
evaluate();
while !is_done() {
  select();
  crossover();
  mutate();
  evaluate();
  survive();
}
```

The other important operation is called *mutation*. It serves to bring new values into the evolutionary process: with some low probability, *e. g.* 0.01, a gene may invert its value.

Figure 1.3 shows a genome before and after mutation.

In pseudocode, a typical GA would look like Listing 1.1.

Evolution Strategies

In ES, as introduced by Rechenberg in 1973 and extended by Schwefel in 1981, candidate solutions to the given optimization problem are usually encoded as fixed-length vectors of floating point values. As elementary components of the genetic material, these floating point values play the same role as the single bits in a GA. Offspring are created by copying an individual and mutating its genes. In contrast to GAs, mutation here means adding some Gaussian noise to the value of a gene. Usually crossover is used only as a secondary operator. Originally it was not used at all.

The evolutionary process is controlled by *strategic parameters*. For example, the variance and the probability of the Gaussian noise regulate the amount of mutation. The strategic parameters may differ among the genes or among the individuals. They may also change during the evolutionary process. The idea behind these parameters is to dynamically control the *evolutionary pressure*, and thus helping the population escape when it is trapped in a local optimum of the fitness landscape.

A variant of ES are *self-adaptive* ES. In these algorithms, the strategic parameters themselves are made subject to evolution by representing them as genes that are attached to the individuals. These additional genes are then evolved in the same way as the other genes. For a survey on self-adaptive ES, see Angeline *et al.* [1].

For ES, several selection schemes along with a special notation have been developed. From μ parents, ν offspring individuals are created. In (μ, ν)-ES, $\nu > \mu$ offspring is generated. None of the parents survive, and the fittest ν individuals of the offspring form the next generation. In $(\mu + \nu)$-ES, parents compete with their offspring. Parents can survive as long as they are fitter than their offspring.

The original application was a $(1 + 1)$-ES. The initial population in this implementation consisted of only one parent individual. From this parent, one offspring was generated. Then the fitness of both was compared, and the fitter one of them

Listing 1.2: Typical evolution strategies algorithm (without crossover)

```
initialize();
evaluate();
while !is_done() {
  select();
  mutate();
  evaluate();
}
```

Listing 1.3: Typical evolutionary programming algorithm

```
initialize();
evaluate();
while !is_done() {
  select();
  mutate();
  evaluate();
  survive();
}
```

survived. In Listing 1.2, a pseudocode example for a typical ES (without crossover) is given.

Evolutionary Programming

In EP, as developed by Fogel in 1966, the representation of candidate solutions is chosen in an especially straightforward, problem-specific way. Genotype and phenotype of an individual are virtually the same in this context. For example, if one wants to find a solution to an instance of the Travelling Salesman Problem (TSP), where the shortest tour that visits each of a given set of cities exactly once is to be found, in EP, each candidate solution would be a permuted list of the cities. For other problems real-valued vectors, graphs, or other suitable structures are used as individuals.

Offspring are created by applying powerful mutation mechanisms to these individuals. Since the mutation can be made flexible enough to introduce great variety in the offspring, recombination is nonessential and in many cases left out in EP, *cf.* Whitley [36, p. 825]. A typical EP pseudocode implementation would look like Listing 1.3.

Genetic Programming

In GP, each individual is a program. The fitness of an individual depends on how well it solves the problem in question. Operations used in GP are crossover and mutation, however mutation is omitted in some implementations. The programs are typically written in a language from the LISP family or another functional language. LISP is especially suitable for this purpose because it permits easy manipulation of tree-like

program code, which is typical for GP. However, there are also GP applications that evolve linear programs or graph-like programs.

GP emerged in the late 1980's. Fujiko and Dickinson [13], de Garis [7], and Koza [19] have worked on this approach. GP has been successfully used for different purposes, *e. g.* to find and evolve heuristics like TSP solvers, particle swarm optimizers, and even other EAs, but also in computer art, financial trade, medicine, bioinformatics, and many other fields. For a tutorial to GP that also contains a large list of resources on GP see Poli *et al.* [23].

1.1.2 Multi-Objective Evolutionary Computing

Many optimization problems aim at several different goals that are in conflict with each other, and thus, it is not possible to find a fully optimized solution that satisfies all goals at the same time. Therefore, one tries to find solutions that cannot be optimized further toward a goal without worsening at least one of the other goals. The problems that share this characteristic are called multiobjective optimization (MOO) problems, which constitute a large research field of their own. When EAs are applied on MOO problems, this is called Multiobjective Optimization using Evolutionary Algorithms (MOEA) or also Evolutionary Multiobjective Optimization (EMOO). For an analysis on MOEA, see Veldhuizen and Lamont [34]. For a short tutorial on MOEA, consult Coello [5]. See also Ghosh and Dehuri [14] for a more recent survey on MOEA.

1.1.3 Interactive Evolutionary Computing

In the types of evolutionary algorithms that have been presented in the preceding sections, once started, the computation runs without human intervention. Reproduction and evaluation are done automatically. The results are usually presented after termination. In contrast, Interactive Evolutionary Computing (IEC) is a sort of EAs in which a human user aids the evolutionary process by providing some kind of feedback on the created candidate solutions. This approach is applicable in cases where it is infeasible or too difficult to devise an appropriate computational fitness function because of a lack of knowledge about what exactly forms a "good" solution.

Types of IEC

Two types of IEC are distinguished, *cf.* Takagi [31, p. 1275]. The differences lie in form and extent of the human intervention.

Narrow definition In IEC of the narrow definition, the user simply provides the fitness values for the generated individuals, instead of having them computed automatically. In other words: the computed fitness function is replaced by a human evaluator.

In some systems that follow this definition, the fitness values may be taken from a small set of typically 4–5 values, while in other systems of this type, values range from 0 to 100. Higher granularity than that is not used as it exceeds

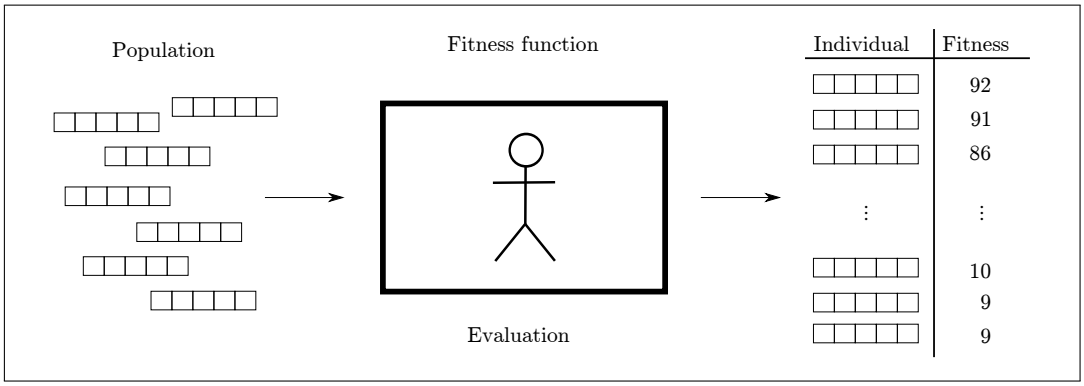


Figure 1.4: Fitness evaluation in IEC. The quality of the individuals is rated by a human evaluator.

the accuracy that human evaluators can reasonably achieve. The evaluation phase is illustrated in Figure 1.4. Compare this to the conventional evaluation in Figure 1.1, where the fitness values are calculated automatically and with higher granularity.

Broad definition In the broad definition, the user intervenes in the evolution process by adjusting mutation rate, freezing the evolution of specific genes, performing selection (*artificial breeding*), *etc.*

Semet [29] provides a literature survey on IEC, putting emphasis on the theory. For a broad application-oriented survey on IEC, see Takagi [31].

Problems in IEC

The introduction of human intervention entails some specific problems. User fatigue has been identified as the most important problem in the field of IEC. As opposed to a machine, the concentrativeness and endurance of a human operator is limited. After some time, users are neither willing nor capable of producing appropriate contributions anymore. Typically, first signs of fatigue show after a few minutes. At the latest after several hours, the point of severe fatigue is inevitably reached; a continuation of the process would be futile.

As an obvious consequence, the amount of user interactions needs to be reduced. Compared to traditional evolutionary algorithms, relatively small populations and only few generations are used in IEC in order not to overburden the user with too many evaluations. While for conventional evolutionary algorithms population sizes of 50 or greater are usual, and thousands of generations are evaluated, in IEC we find typical numbers of at most 10–20 individuals in a population and at most 20–25 generations, leading to at most $20 \cdot 25 = 500$ evaluations, *cf.* Takagi [31, pp. 1276–1277]. In order to compensate for a decreased number of generations, the convergence speed must be increased. There are several means to do this. One method uses an

initial population that is derived from the user's basic preferences instead of being just random. Another method tries to accelerate the convergence by synthesizing an optimal extra individual in each iteration, combining the best features of all the individuals of that generation. A drawback of accelerating the convergence is that it might lead to *premature convergence*, *i. e.* the evolutionary process more likely arrives at a suboptimal solution.

Various approaches have been proposed to overcome the problem of user fatigue, see for example Llorà *et al.* [20], Kamalian *et al.* [18, 17] (ranking solutions by promotion, demotion and neutral; combination of computed fitness evaluation and sporadic human interaction), Sáez *et al.* [28] (reference chromosome, modified crossover operator with independent probabilities according to user's selections). Pallez *et al.* [22] even use an eye-tracking device to help minimize user interactions.

Another problem that is inherent to IEC is the problem of inaccurate and disproportionate ratings. Users apparently have only a vague understanding of where a candidate solution should be placed on an absolute scale. While they often manage to indicate the right qualitative tendency, the exact amount they state seems to be arbitrary.

Most notably, users tend to assign extreme ratings to candidate solutions that show some feature they have a strong opinion on. An examination of these ratings often reveals that they are exaggerated and incommensurate to other ratings the users have given. Particularly, unjustified low ratings have been observed, *cf.* Kamalian *et al.* [18]. Even if most of the features of a candidate solution are acceptable to them, users not necessarily assign a moderate rating to it.

The aforementioned problem is worsened by unsteady preferences: Users might change their opinion over time. This means that the ratings they have already provided might become even less appropriate.

This lack of appropriate ratings makes it difficult to establish a ranking of the solutions that properly reflects the users' actual preferences. In order to deal with this problem, it has been proposed to use rating systems with low fitness granularity, see Semet [29, p. 6].

1.2 Layout and Meta Layout

In order to comprehend the following introduction to layout and meta layout, it is important to know the basic terminology of graph theory. We assume here that the reader is familiar with it.

Graphical modeling is a technique widely used in many fields, *e. g.* in Model-Driven Software Development (MDSD). In graphical modeling, system models are represented using some kind of graphical notation. These models have to be thought of as abstract concepts that are usually based on graphs in the broadest sense. In order to support humans who want to grasp the meaning of a model, talk about or modify it, it is crucial to present the model in some way that aids human consumption. Usually, this is accomplished by drawing a two-dimensional representation of

the underlying graph, *e. g.* onto a screen or a chalkboard. Such a visual representation is called a *drawing* of the graph. Nodes are typically drawn as circular or rectangular shapes. Edges are drawn as sequences of connected straight lines, or as curves. Nodes and edges may have captions that are represented as text. Further constraints may be imposed on the drawing, *e. g.*

- directed edges shall follow a preferential direction,
- connected nodes shall be drawn close to each other, and
- nodes and/or edges shall be placed on a grid.

There is no unique drawing for a graph. On the contrary, for a given graph, virtually infinitely many drawings are possible. However, not always can all the constraints be fulfilled simultaneously. Moreover, depending on the domain, on the use-case scenario, and on the personal preferences of the user group, some drawings are more suitable than others.

Therefore, the goal is to find a suitable *layout*, *i. e.* a suitable arrangement of the elements in the drawing. A *layout algorithm* is an algorithm that is used to calculate such a layout. Thus it relieves the user from the burden of doing this manually.

Many different layout algorithms for different types of graphs and graphical models have been devised. For a bibliography of layout algorithms, see di Battista *et al.* [8]. Some of these layout algorithms have parameters, *e. g.* desired layout direction, minimum node distance, and desired aspect ratio. These parameters serve for customizing an algorithm without changing its implementation.

Dependent on their underlying approach, the layout algorithms can be grouped into different *layout types*, *e. g.* circular, layered, force-directed, or orthogonal. Note that there are also layout algorithms that do not primarily aim to facilitate human consumption, *e. g.* in integrated circuit (IC) design. However, these are beyond the scope of this thesis. Furthermore, for some diagram types, such as sequence diagrams, there are rather strict rules for layout, which means that mapping those diagrams on a graph-like structure for layout is too much of a detour. More direct layout approaches are needed to produce layout for them which are beyond the scope of this thesis as well.

Practical aspects of handling graphical system models have been addressed by Fuhrmann and von Hanxleden [11]. They propose to apply the Model-View-Controller (MVC) architectural pattern to the pragmatics of model-based system design. Speaking in the terminology of the MVC concept, a concrete drawing of a given model is called a *view* on the model.

The meta layout infrastructure that is integrated in KIELER provides an automatic layout service as an aid in the handling of graphical models with KIELER.

According to Fuhrmann and von Hanxleden [12], the "idea of *meta layout* is to synthesize views automatically, thus freeing the user to focus on the model itself."

As Fuhrmann and von Hanxleden point out, the state of the practice in devising a graphical model is to manually create a single view of it using a What You See Is

What You Get (WYSIWYG) editor. This view is from then on used to inspect and modify the model.

In contrast to this widely-used approach, the meta-layout approach presented by Fuhrmann and von Hanxleden more clearly separates the model from the view. This makes it possible to synthesize several views of the same model by applying different layout algorithms to the model.

One could say meta layout affects the layout on an abstract level. This comprises the choice of an appropriate layout algorithm from a set of already existing layout algorithms and the specification of parameters and constraints for this layout algorithm.

1.3 Aesthetics metrics

This section explains the concept of aesthetics metrics, which are used in order to measure and compare the quality of different graph layouts.

In the design of layout algorithms, the question is "How can the given graph be drawn so that users find it appealing and understand its meaning?"

Every layout algorithm that is designed to facilitate human consumption needs to produce layouts that adhere to some *aesthetic criteria*. These are rules that specify how the layout shall be in order to look "appealing". For example, "Edge lengths should be uniform" is an aesthetic criterion, another is "Nodes should not be too close together". It is important to know that different aesthetics criteria can conflict each other. Therefore, it is difficult or even impossible to find layouts that adhere perfectly to different criteria at the same time. Rather, one needs to trade one criterion against another. Therefore, it is necessary to find out which criteria are the most useful ones for human understanding. Purchase *et al.* [24] have performed empirical studies on the effect of various layout aesthetics on human understanding of graphs. They found a significant detrimental influence of edge bends and edge crossings on the understandability of graphs. Völcker [35] has studied how different aesthetics metrics influence the preference and understanding of graphical diagrams. His analysis was restricted to Statecharts, a certain type of graphical diagrams, and was conducted using Kiel Integrated Environment for Layout (KIEL), the predecessor of KIELER. For more on layout aesthetics, see *e. g.* Coleman and Parker [6].

In order to measure the extent to which a drawing conforms to an aesthetic criterion, one can define an *aesthetics metric*. An *aesthetics metric* expresses the presence of an aesthetic in a drawing as a non-negative number. This way, it makes any two drawings comparable with respect to the criterion. For example, if we have two drawings D and D', and an aesthetics metric $\aleph$ with $\aleph(D) > \aleph(D')$, then D is superior to D' with respect to $\aleph$.

Purchase [25] presents continuous formal metrics for measuring the aesthetic presence in a general graph drawing for seven aesthetic criteria. With continuous metrics, the extent to which an aesthetic is present in a drawing can be conveyed. The met-

rics proposed by Purchase are scaled to lie between 0 and 1, in order to make the metric values independent of the nature of the underlying graphs.

More recently, Bennett *et al.* [2] give a survey on the research in the area of aesthetic heuristics.

1.4 Used Technologies

This section lists the technologies which EVOL makes use of.

Eclipse Eclipse[1] is an open-source extensible integrated development environment (IDE) written in Java. Extensions which contribute functionality to Eclipse may be added via plug-ins and an extension-point mechanism.

Kiel Integrated Environment for Layout Eclipse Rich Client KIELER is a research project about enhancing the graphical model-based design of complex systems that is developed by the Real-Time and Embedded Systems Group[2] of the University of Kiel.

KIELER Infrastructure for Meta Layout KIML[3] is a subproject of KIELER that deals with the automatic layout of graphical models.

Following the concept of meta layout as explained in Section 1.2, KIML manages the use of different layout algorithms, which are classified in KIML by their underlying approach, (*e. g.* layered, force, *etc.*). Graphical diagrams are classified by their type, *e. g.* class diagram, state machine, or data flow diagram.

When automatic layout is requested for a specific diagram or diagram part, KIML decides which layout algorithm to use on this item and how to configure the algorithm. Then KIML delegates the layout task to the algorithm. The result of the automatic layout process is an instance of a data structure called *layout graph*. According to the KIML project website[3], a layout graph is an "internal representation of the graph structure of the current diagram, built on the KGraph model." This layout graph contains concrete coordinates for the diagram elements. In a final step, the layout is imposed on the diagram, *i. e.* the coordinates from the layout graph are assigned to the corresponding diagram elements.

KIELER Graph Analysis (GrAna) GrAna is a KIELER plug-in that provides a framework to analyze graphs and graph diagrams, given as *KGraphs* (graph representation with hierarchy) as defined by the KIELER core. GrAna comes with a set of predefined analysis providers, which can be used to determine properties of the graph such as node count, edge count and node degree, but also properties of a specific drawing of the graph, such as edge crossings count, average edge length and many more.

[1] http://www.eclipse.org/
[2] http://www.informatik.uni-kiel.de/rtsys/
[3] http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KIML

Further analysis providers may be added to GrAna via the Eclipse extension-point mechanism. Thus, GrAna can be used to implement aesthetics metrics based on diagram analyses.

For a detailed description of GrAna, see Rieß [26].

1.5 Related Work

Various approaches have been proposed that use EAs for graph layout. In these approaches, an individual typically either represents the graph structure, or a list of node positions. Given a graph as input, the algorithm is applied in order to find a "good" concrete layout for this graph. The understanding of "good" is typically defined implicitly in the fitness function.

Some of the approaches solve rather specific layout problems, *i. e.* they target specific graph types or specific aesthetics. For example, von Gudenberg *et al.* [16] have implemented an EA for the layout of Unified Modeling Language (UML) class diagrams. Groves *et al.* [15] experiment with GAs to draw directed graphs. Their specific goal is to minimize edge crossings and upward edges. Utech *et al.* [33] have used an EA to optimize the layering and the ordering of directed acyclic graphs.

Other authors are aimed at more general solutions, *i. e.* solutions which are more independent of specific graph types or aesthetics. Rosete *et al.* [27] develop a general approach for Graph Drawing, using $(1 + 1)$-ES. Tettamanzi [32] presents an algorithm for drawing undirected graphs according to a number of aesthetic criteria. Coleman and Parker [6] view the task of graph layout as a MOO. Based on a general aesthetics function that is the composition of one or more possibly conflicting numerical layout aesthetics, they present the "Aesthetic Graph Layout (AGLO)", a method which serves to optimize this aesthetics function. They conclude that their layout method is "general, flexible and uniform", and moreover "provides a means for trading off between conflicting aesthetics".

Some authors ask for user interaction in order to take personal preferences into account. Biedl *et al.* [3] propose a "multidrawing" approach for graphs. Their implementation uses a spring force layout algorithm which produces a number of drawings, emphasizing inherent symmetry of a given graph. These drawings are presented simultaneously to the user, so that he or she may pick the "best" one of the generated layouts. Masui [21] uses GP to learn layout constraints from "good" and "bad" layout examples presented by the user. Do Nascimento *et al.* [10] present an interactive GA for directed graph drawing. The user may contribute by providing hints, such as indicating which region shall be improved primarily, imposing constraints for node orders, or manually setting the position of nodes.

As opposed to the cited approaches, the work presented here applies optimizations not on concrete layouts, but on a more abstract level, which includes independence of concrete diagram types and layout algorithms.

Based on the idea that already many layout algorithms exist that are capable of producing nice layouts, and which just need to be picked and configured properly in

order to comply to requirements set by use-case scenario or by specific user preferences, this approach focuses on finding out how their configuration can be optimized in order to produce better layouts. An objective analysis of concrete diagram layouts based on continuous metrics, as well as interactive subjective rating of concrete layouts are part of the concept.

2 KIML-EVOL: Evolutionary Meta Layout

This chapter provides a deeper insight into the problem statement and proposes an approach to the topic of evolutionary meta layout.

First, we need to take a look at the relevant parts of KIML, since this project is intended to be an add-on to KIML.

KIML manages a set of entities that are called *layout providers*. Each layout provider can perform one or more layout algorithms on the child nodes of a given node in a graphical diagram editor. The layout providers are added to KIML by extending a certain abstract class and using the *layoutProviders* extension point. They are classified into different *layout types*, depending on the approach of the layout algorithms, *e. g.* force-directed, hierarchical or circular. The list of layout types used in KIML is extensible.

One may request automatic layout for a given diagram or diagram part. This can be done via a user interface (UI) or programmatically. Moreover, KIML allows the user to specify *layout options* for the parts of the diagram on which the layout process shall be performed. These layout options affect the automatic layout process. Thus, they enable the users to express their preferences in form of adjustments of the default automatic layout. Some of the layout options are defined by KIML, while others are contributed by the layout providers. A layout option consists of a unique identifier, a data type and a target specification. The identifier serves to identify and discern the layout options from each other. The data type specifies the type of the value the option can have. In the current implementation, KIML distinguishes between boolean, integer, floating-point, string, and enumeration valued options. The target specification indicates which targets are affected by the option. A layout option may apply to either nodes, ports, edges, labels, or entire diagrams. While some of the options apply only to certain algorithms or certain diagram types, others apply to general graphs, regardless of diagram type and layout algorithm.

A set of layout options is called a *layout configuration*. A layout configuration can be used to request values for the options that it contains. The values are either assigned beforehand or calculated on demand. A specific layout configuration used to calculate a layout for a specific diagram usually produces the same layout result every time it is done, *i. e.* the layout is reproducible.

We can use this to build a system that uses an EA in order to explore the meta-layout search space by evolving layout configurations.

First of all, we need a genetic representation of meta layout information. For this, we need to draw a line between what shall be evolved and what shall be part of the

environment. A question that needs to be answered is whether the diagram should be a fixed part of the environment, or whether it should be subject to evolution.

The author finds it reasonable to fix a specific diagram as a part of the environment. On the other hand it could also be of interest to encode the diagram or diagram type, *e. g.* to find a suitable diagram for a fixed layout configuration. However, the actual benefit thereof seems rather questionable.

Therefore, we focus on evolving layout configurations for a given diagram. We only need to define a genetic encoding of layout configurations and an evolutionary process that can be used to evolve them.

This is based on the assumption that specific layout configurations lead to reproducible layout when used on a specific diagram. However, this is not the case for all layout algorithms: some layout algorithms heavily depend on pseudorandom values. For example, the Fruchterman-Reingold algorithm places nodes on pseudorandom locations somewhere near their original positions. The pseudorandom values used in these algorithms are generated internally, based on some random seed number s, which may or may not be settable by the user. For a fixed s, the generated sequence of pseudorandom numbers will be the same every time the algorithm is run, resulting in the same layout, while for a different s, a different sequence will be generated in each run, resulting in a different layout. This means that for the same input graph and layout option settings, but a different s, the resulting layout is not necessarily identical. Five layouts of a small graph are shown in Figure 2.1 to exemplify this. Even though the differences among them are apparent, they have been generated using exactly the same layout configuration.

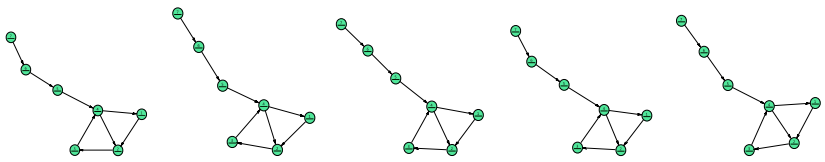


Figure 2.1: Different results of a randomized layout algorithm using the same options

In some cases the differences between two runs using the same settings can be even bigger than those between two runs using slightly different settings. This states a problem for optimizers that depend on a fitness function—as do EAs—, for it adds s as another parameter to the fitness function.

In order to preserve the effectiveness of the optimizer and prevent it from being disoriented, it is favorable to use a fixed value of s. If s cannot be controlled by the caller, then there is in effect no proper fitness function, but only something like a fitness relation that yields irreproducible values. Therefore, we advocate that randomized layout algorithms be designed in a way that does not allow the results to vary wildly. Despite the randomness, they should at least remain predictable to a certain extent. For "rampant" layout algorithms, we can expect the optimization process to be less effective. However, it is unclear how serious this problem is in

practice. Dealing with this exceptional case is subject to future work.

So far we have described the existing elements of the KIML framework we want to use; now how does user feedback come into play? In order to enable the user to interactively provide feedback, a UI is needed. The UI should display created layout proposals in an appropriate fashion. It seems reasonable that this should be in a graphical way, much like how conventional automatic layout is done in KIML. Furthermore, the UI should enable the user to rate these proposals. It is common practice to provide such an input by buttons, sliders, text boxes, or similar controls. For this purpose, an appropriate means could be a pair of buttons to promote or demote a proposal. Another suitable way could be a slider that allows one to set a value between 0 and 100%.

Since user fatigue is the main problem in IEC, we should find a way to reduce the amount of user interaction. We assume that there is a relation between the measurable aesthetic features of diagrams and the user's subjective concept of layout quality. If we used a set of aesthetics metrics and if we knew the relative influence of the metrics on the user's opinion from ratings the user has already provided, then we could try to predict the ratings that the user would assign to newly created layouts. If our predicted ratings match the user's opinion, then the user would not need to provide so many ratings. Thus, a rating prediction system could spare the user a lot of interaction.

In summary, the problem statement is to build a system consisting of an EA framework that is used to evolve layout configurations, a user interface that is used to collect user feedback, and a subsystem for rating prediction.

This section has elaborated the problem statement. Given the aforementioned requirements and the technologies listed in Section 1.4, how could a solution look like? An approach to solve this question is proposed in the following sections. The presented approach to evolutionary meta layout has the full name KIML-EVOL, from now on called EVOL. The following section explains the concepts and design decisions for EVOL. The UI that had to be implemented so that user feedback can be collected is presented in Chapter 4.

2.1 Representation of Data

The basic idea of the presented approach is to model layout options as genes. An implementation of an EA produces an initial population of layout option genomes. Each individual of this population is a candidate solution, consisting of a list of genes. For each layout option that is to be learnt, every individual has exactly one gene. In each gene, a value for the respective option is encoded.

Genes For the design of genes, we can consider a gene as a kind of container that holds a value. The properties of this value depend on the layout option. The set of valid assignments differs from one option to another. For example, a layout option called *layout direction* could accept one of the following values: *up*, *down*, *left* and

right. Any other values are invalid for this option. A gene that encodes a value for this specific option must adhere to this restriction, that means its value must be equal to one of the above-mentioned values, otherwise it would render the solution infeasible. This restriction not only applies to this gene, but it propagates to all copies and variations of the gene that result from evolutionary operations. This means that evolutionary operations that work on gene level have to regard the types of the genes.

Individuals As stated before, an individual is modeled as a list of genes, one gene for each layout option that is to be learnt. Since this set of layout options may vary, it is reasonable to allow individuals of arbitrary size. Furthermore, one needs to answer the question whether individuals should consist merely of their genetic information, which would make them rather plain and "light-weight", or whether an individual should be enriched with additional properties, *e. g.* its rating. In the latter case, one would not need external data structures to store additional information about the individuals. Everything could be stored at hand, in the individuals themselves.

Populations A population should be seen as a list or a set of individuals. A question that should be answered is whether this should be mutable after creation. As we have seen in Section 1.1, the population size in an IEC should be limited to not more than 20 individuals, otherwise we would provoke user fatigue. A population size of about 10 individuals makes it possible to display meaningful information about each individual on the screen at the same time.

2.2 Evaluation of Individuals

Having established what constitutes an individual, the question arises how such an individual should be evaluated in order to determine its fitness.

The values the individual encodes for the layout options need to be used as parameter values for a layout algorithm. This is done by synthesizing a layout configuration. The choice of the layout algorithm itself may also be encoded in the individual.

The layout algorithm is subsequently applied to the diagram that is opened in the current editor. These steps are performed using methods that are provided by KIML. From each application of the layout algorithm we retrieve a layout proposal in form of a layout graph. The layout graph is used to impose its layout information on the diagram, which gives a new drawing of the underlying model. This drawing can be considered as the phenotype of the individual. The phenotype embodies how an individual appears to the user. Since this appearance is the target of our optimization, the numerical fitness of the respective individual must be derived from the phenotype. However, one must keep in mind that the same genome, applied to a different diagram, could have a very different appearance. For example, some individuals might produce very appealing layout results on small diagrams, but messy

results on large, complicated diagrams. Therefore, the optimizations achieved on a certain diagram are not necessarily generalizable.

Automatic Measurement In the previously described step, a layout algorithm produced a layout proposal in form of a layout graph using the layout configuration encoded in an individual. This layout graph relates to both the individual and the diagram it was targeted at. The layout graph is subsequently analyzed by performing an automatic analysis using GrAna, according to some continuous aesthetics metrics which follow the concept of described in Section 1.3.

The measurement process yields a map of aesthetic features and corresponding measurement results. Each result is a real number that is scaled to lie in the interval of $[0, 1]$. The value conveys the extent to which a certain aesthetic criterion is present in the layout. If the layout conforms perfectly to a criterion, then the respective value is 1. In contrast, if the layout does not conform to a criterion at all, then the value is 0.

Automatic Rating After automatic measurement has produced measurement results for a drawing, these measurement results form the basis of *automatic rating*.

The concept of automatic rating is to automatically predict the fitness of an individual. Note that while the actual fitness is provided by the user and thus dependent on the user's personal preferences, automatic rating might support the user by predicting the rating. The more accurately the prediction conveys the user's preferences, the less work the user has to do. This is based on the assumption that the user's preferences somehow relate to the perceivable features of the drawing that can be measured by an automatic diagram analysis.

One way to obtain an automatic rating from the measurements is the following.

Let k be the number of measured features, and $m = (m_1, m_2, \ldots, m_k)$ be the measurement results. To let each measurement contribute a specific amount to the rating, the weighted sum of the results is calculated. We use a vector of positive constants $w = (w_1, w_2, \ldots, w_k)$ as weights, and define an automatic rating proposal

$$r = \sum_{i=1}^{k} w_i m_i \ .$$

The assignment of w determines how important each measured feature is to the user. This method is called *additive composition*, *cf.* Coleman and Parker [6].

User Rating Each time the EA has produced a new population, it is the user's turn to rate the new individuals, depending on the opinion he or she has on the respective layout. The UI needs to provide suitable methods for assigning ratings to individuals. As the automatic rating prediction also provides rating proposals for all individuals, it is not necessary to rate each individual manually. If the user provides a rating for an individual, this overrides the value proposed by the rating prediction. Otherwise,

the proposed value is accepted as rating for the individual. The ratings serve as actual fitness values for the individuals.

2.3 Evolutionary Process

The evolutionary framework implemented here tries to separate the evolutionary operations and the concrete implementation, not polluting the framework with too many problem-specific details. On the other hand, the amount of abstraction should be restricted to keep usage and maintenance manageable.

The compromise that has been found here makes it possible to use the framework both for the evolution of layout options and for the evolution of rating prediction without rendering it impractical by making it overly complicated.

Initialization Every evolutionary process begins with an initial population. A new population is created based on the layout configuration of the current diagram view.

Recombination and Mutation EVOL makes use of two genetic operators that generate new offspring: recombination and mutation.

Mutation is a means to bring new values into the evolutionary process. In EVOL, we must regard the different types of genes. For example, a gene encoding a floating-point valued option would be mutated differently from a gene encoding an enumeration valued option.

As stated in Section 1.1, recombination (often called *crossover*) of two or more parent individuals creates a new genome based on the genes of the parents. Note that the individuals must belong to the same problem instance, *i.e.* they must encode the same options in the same canonical order.

Again, the different types of genes must be taken into account, because different recombination schemes are possible for different types. The most simple way is just taking the values from the parents and deciding in a random fashion which of the parents' values to use. This approach, as exemplified in Figure 2.2(a), works for all types of genes.

For integer and floating-point valued genes it is possible to calculate an average of the parents' values, and put this into the offspring individual. If this method is used consistently for all genes, the offspring can be considered as being on the common barycenter of its parents. See Figure 2.2(b) for an example.

Selection The next question is which selection mechanism is appropriate for the evolutionary process in EVOL. Should we use a ranking selection, in which the best k individuals of the population are granted an equal chance of producing offspring, or should fitness-proportional selection be used, *i.e.* the chance of producing offspring is proportional to the fitness?

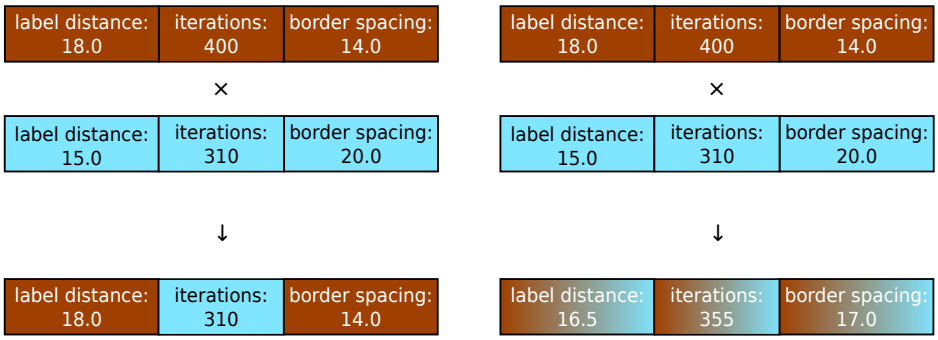


Figure 2.2: Examples for two different types of recombination:
(a) Recombination by simple crossover (b) Recombination using average values

The ranking selection mechanism, which is easiest to implement, strictly prevents reproduction of the individuals with the lowest ratings. It relies on the accuracy of the coarse tendency of the given ratings.

For fitness-proportional selection, the accuracy of the relation between the ratings in the population is more important. When all ratings are similar, even the individuals with lowest ratings are conceded a fair chance of reproducing.

For the purpose of this project, ranking selection appears more suitable, because the fitness values are not too reliable, and we would not want to lose diversity because of inappropriate fitness values. By this choice, we accept that the individuals with lowest ratings in a population are never allowed to reproduce.

Diversity As evolution goes on, the populations might become more uniform, *i. e.* most of the genomes share equal or similar genes. This is because random mutations applied to high-fitness individuals are more likely to produce inferior individuals which do not reproduce. Thus, advantageous mutations become more rare. Crossover of similar genomes produces even more similar genomes.

The lack of diversity bears the danger of not exploiting the solution space extensively enough, leading to premature convergence. There might be better solutions that are missed, because they are very different from the genomes in the current population. Furthermore, having too many similar genomes might provoke user fatigue. Monotony can be very tiring to humans and should therefore be avoided. Thus, it seems reasonable to check whether the population contains genomes that are "too similar" to each other. Only one representative of a group of similar genomes needs to be retained. The others may be discarded.

But how do we know when two genomes are too similar? We need a distance function for genomes, a *distance metric*.

Distance Metric for Genomes When designing a distance metric for genomes, the type of the genes needs to be considered, as in the design of mutation and recombination operators. Typically, slightly changing the value of an integer valued option does not affect the layout as much as does the variation of an enumeration valued option. For example, changing the layout option *minimum node distance* from 60 to 58 allows the algorithm to subtly place the nodes a little closer together, but changing the *layout direction* from *right* to *up* causes the whole diagram to turn 90 degrees counterclockwise. Provided the diagram is rotationally asymmetric, the differences concerning the node positions and edge directions should catch even an extremely unheeding user's eye, though except for the rotation, the layout is actually the same.

On the other hand, if one increased the minimum node distance by a larger amount, say from 60 to 4000, the effect on the layout would possibly be stronger than the effect of changing the layout direction. Somewhere in between, there is a fuzzy range in which both changes appear equally strong.

Of course this cannot be said in general, as it would be easy to devise a layout algorithm that shows arbitrary behavior. Layout options can interfere with each other, amplifying or weakening each other. In some cases, a slight change of a single option can have an enormous effect on the layout, while a moderate variation of 5 other options leads to no conceivable difference. This may be a flaw in the layout algorithm, but it may also be intentional. For example, there may be a boolean option that enables or disables the effectiveness of some other options. Moreover, the effectiveness of layout options may depend on properties of the graph for which layout is calculated. For example, if the graph consists only of isolated nodes, any options targeting the layout of edges would presumably have no observable effect, since there are no edges. If on the other hand the graph had higher density, the same options related to edge layout might well have a considerable effect on the drawing.

All in all, one should not make too strict assumptions on how strong the changing of an option would affect the layout. The assumptions we make here are the following.

- The more genes differ in a pair of genomes, the higher their distance should be.
- A relatively large change of a floating-point valued option should imply a higher distance than a merely slight change.
- The change of the layout algorithm should mean a large distance.
- A difference in the layout type of the algorithms should have even more impact on the distance.

An alternative approach would be to compare the phenotypes—*i.e.* the actual drawings—instead of the genes of the individuals. The advantage would be that it could ensure that the drawings which are presented to the user always appear different from each other. This might help to prevent user fatigue. The disadvantages are that firstly this approach would require much higher computational effort, and

secondly it might lead to premature convergence, since the genotype space is likely to be not exploited extensively enough. For these reasons, in the first implementation of EVOL we refrain from comparing the drawings and stick to comparing the genes.

However, another interesting approach could be to compare the measurable features of the drawings. Drawings that differ in many features should be considered more different as compared to drawings that differ only in few features. Note that a user might find two drawings equally appealing even though they differ in many features. Although these features are usually aggregated values and therefore can cover merely a subset of the possible differences between two drawings, the amount of differences they reveal should be sufficient in practice. Since these features are analyzed anyway, the computational overhead introduced by this approach would only come from the comparison of the features, which—except for very small graphs—is far less costly than the comparison of the entire drawings.

2.4 Evolutionary Rating Predictors

In Section 2.2, the concept of automatic rating was introduced. The automatic rating for an individual was obtained by calculating the weighted sum of the measured features of the respective drawing. But where do the weights come from that are needed for this approach? They depend on the user's personal preferences for the domain, and thus cannot be known in advance. However, one could try to learn them from the feedback the user provides on the drawings.

EVOL implements an approach that tries to learn appropriate weights by using an EA. It makes use of the ratings the user provides on the graph drawings. The algorithm evolves a population of *rating predictors*. Each rating predictor is a genome consisting of *weight genes*. Each weight gene encodes a weight, which is a floating-point value. From each rating predictor an automatic rating can be calculated using the the weights encoded by the predictor. In most cases, different rating predictors will yield different ratings.

We denote the mismatch between the rating r_p that is predicted by a rating predictor p and the actual rating r_u that is given by the user as the *prediction error* $e_p = r_p - r_u$.

The prediction error can be positive or negative, depending on whether the predicted rating was too high or too low. However, this is irrelevant when we want to determine the quality of the predictor, so we consider the absolute value $|e_p|$ of the error. For a "good" predictor this would be a small value, and for a "bad" predictor it would be a big value, so the goal here is to minimize $|e_p|$.

For a fitness function, according to convention (see Section 1.1), it should be *vice versa*: the better the predictor, the higher its fitness value. Nonetheless, we can use $|e_p|$ to determine the fitness by subtracting $|e_p|$ from a constant value M that is big enough that the fitness value is always positive.

Thus, we define the fitness function by setting $f(p) := M - |e_p|$.

2.5 Handling of Multiple Diagrams

As stated before, the evaluation of individuals on a single diagram is not necessarily generalizable. To address this problem, one could try to apply each individual to a set of several different diagrams. In Eclipse, it is possible to open several diagrams at the same time, using several diagram editor windows. It is also not too difficult to extend the application of the layout algorithm to several diagram editor windows. However, this *multi-editor approach* is nonetheless tricky to implement, as the devil is in the details. For example, one needs to answer the question how to determine which diagram editor windows should be affected. What should happen if the user decides to minimize or close some of the editor windows? What if he or she opens new ones? A straightforward solution would be to dynamically find out which diagram editor windows are visible every time an individual is applied, and calculate layout for all of them successively or concurrently. Another solution could incorporate a diagram viewers list which controls for which diagrams layout should be calculated. The user would be required to add and remove diagram viewers from the list. Configurable predefined sets of viewers could assist in this task.

Another question is that of zoom level, size, and position of the different diagram viewer windows. Should the zoom level be coupled in all the windows, or should the user be allowed to set it differently for each window? How should the windows be sized and arranged? Should we give the user the freedom to arrange the windows as he or she likes, or would it be better to arrange them automatically in a canonical way, *e. g.* in tiles of the same size?

Yet another question is how multi-editor evaluation and automatic rating should interact. If we allow each individual to perform in several editor windows, there is no more a one-to-one relationship from the rating of an individual to its phenotype in an editor, but now each individual appears in multiple editors. Clearly, the rating of an individual should cover all its appearances. But should it be in equal measure, or should it for example be weighted according to the size of the graphs on which the diagrams are based?

Moreover, the automatic rating would need diagram analysis results from all the diagrams. Then the next question is, where should these results be collected? Should the results from all the diagrams be aggregated somehow before providing them to the rating predictors, or should the rating prediction treat each diagram individually, and finally the total rating would be combined from the ratings of each diagram? While the latter approach seems more reasonable at first sight, the former one might possibly produce similar results while having better computational complexity.

Then, how should the ratings provided by the user be fed back to the rating predictors in order to train them? How can the user ratings be related to the diagram analysis results from the different diagrams?

Finding a solution that properly combines multi-diagram evaluation and adequate automatic rating is quite complex and goes beyond the scope of this work.

Apart from these above-mentioned conceptual problems, the multi-diagram approach also bears the drawback that it makes the evaluation task more difficult and

cumbersome for the user, as he or she needs to rate several diagrams at the same time.

2.6 Saving and Loading Evolution Configurations

The results of evolutionary meta layout are evolution configurations which can be used ad hoc, *i.e.* applied to the diagram in the current editor. This modification can be made persistent by saving the diagram. If the user saves the diagram, KIML stores the layout configuration persistently in the notation model of the diagram. This way, the layout configuration is linked to the diagram.

Note that the particular configuration has been learnt from the preferences of the user who has provided feedback on the layout for a specific diagram, so it is not only related to the diagram and the underlying graph, but also to the particular user.

Based on the assumption that this configuration can be of value for other users or other graphs or other diagrams, one could think of a generalization process which would make it possible to transfer the configuration.

In order to transfer the results, one needs a way to store the configurations permanently, but independent of a certain diagram. Therefore, one needs to answer the question how configurations should be linked with higher-level targets. Higher-level targets could be editor identifiers, diagram types, specific diagrams or models. A rather technical question is the link direction: Should the configuration carry a link to its associated targets, or the other way round, or both? Concerning the link target, a first analysis of the possibilities gives some answers.

- Linking with editor identifier: If we take multi-editor mode into account, where the same configuration can be used for several editors at the same time, this means we would have to either associate a collection of several editor identifiers to each configuration, or independently associate the same configuration to several editor identifiers.

- Linking with diagram type: It seems reasonable that one would not want the same configuration for all diagram types, but possibly a specific configuration can be used for several types. It is presumably a good idea if the configuration "knew" to which diagram types it may be applied.

- Linking to specific diagram or model: To make configurations useful, this should be implemented in a way that makes it possible to link each configuration to several diagrams.

It is conceivable that the usefulness of each of the above-mentioned links may vary according to the circumstances. Therefore, high flexibility is advisable if one wants to persist and transfer layout configurations. In this work, a preliminary solution is included: EVOL offers a method for storing the layout configuration of the current individual in a text file. Developing a more sophisticated approach would possibly

imply to enhance the modeling and management of layout configurations in KIML. Further work on this is necessary.

3 Implementation

This chapter describes the implementation of EVOL. Firstly the structure of the implementation is introduced, and secondly the plug-in interface is described. The last two parts of this section cover how automatic measurement and multi-diagram mode are implemented in EVOL.

3.1 Structure

EVOL consists of six Java packages, which are described in the following paragraphs. For the sake of brevity, the prefix "`de.cau.cs.kieler.kiml.`" of the package names is omitted.

`evol` The core classes of the plug-in are defined in this package. The computational model of EVOL is contained in the class `EvolModel`. An instance of this class models the data that are intended to be displayed in the Evolution view. The model basically consists of an instance of an evolutionary algorithm, a population, and a current individual. Additionally it manages a population of rating predictors that is evolved separately.

`evol.genetics` This package contains a part of the evolutionary framework. Essentially, it defines the data structures for genes, genomes, and populations.

`evol.alg` This package provides an abstract evolutionary algorithm and a basic concrete implementation of an evolutionary algorithm. This implementation uses the evolutionary data structures defined in the previous package.

`evol.ui` This package contains the UI classes, among which `EvolView` is the most important, because it implements the main component of the UI, the Evolution view.

`evol.handlers` In this package, command handlers are implemented, which extend `AbstractHandler` (from the Eclipse package `org.eclipse.core.commands`), so they can be used as default handlers to commands as specified by the `org.eclipse.ui.commands` extension point. These handlers define behavior to manipulate the evolution model. The handlers can be triggered for instance by clicking on the respective command buttons in the view.

- The `ResetHandler` performs a reset of the model, *i.e.* a new population is created and the evolutionary algorithm is started.

- The `ChangeRatingHandler` serves the purpose of allowing the user to promote or demote the currently selected individual from the model by assigning a new rating to it. In the presented implementation, this handler uses a parameter called `...evol.amount`, which specifies the value that shall be added to the current rating. The value may be negative. Thus, both the Favor and the Disregard button trigger this command, but with a different value for the amount.
- The `AutoRateAllHandler` is a command for performing automatic rating on the current population. It successively computes a rating proposal for each individual, based on the layout that is calculated.
- Finally, the `EvolveHandler` serves the purpose of performing a step of the evolutionary algorithm, thus proceeding to the next generation of layout option genomes.

`evol.metrics` This package provides seven aesthetics metrics and some auxiliary analyses that all implement the interface `IAnalysis` which is defined by GrAna. This means they can be added as *graph analyses* for GrAna via the corresponding extension point. The property *category* of the extension element must be set to `...evol.metricCategory` for each metric, so that EVOL can discern metrics and auxiliary analyses.

The only method these classes exhibit is an implementation of `doAnalysis`, which in this case returns a floating-point value.

3.2 Extension Point for Evolution Data

EVOL defines an extension point named *evolutionData* that allows one to specify which layout options can be learnt, plus how they should be mutated. The probability distribution may be either Gaussian or uniform. This extension point references the KIML layout options defined by `...layoutProviders`. This means that the layout options need to be defined in KIML beforehand.

3.3 Automatic Measurement

EVOL uses GrAna to measure aesthetics of graph drawings. For this purpose, EVOL adds some special analysis providers to GrAna via the Eclipse extension-point mechanism. These special analysis providers defined by EVOL are called *metrics*. While GrAna allows analysis providers to produce results of arbitrary type and complexity, the metrics have in common that their result is a simple `Float` object. This `Float` object is normalized so that its value is within the interval of $[0, 1]$.

In the following, we take a closer look on the setup of such a metric.

Listing 3.1 exemplifies how a typical metric is implemented. The interface that is to be implemented is `IAnalysis` from the GrAna package, which defines a public

Listing 3.1: Typical analysis provider

```
// import declarations ommitted for brevity

/**
 * Identifier for "edge count analysis".
 */
private static final String GRANA_EDGE_COUNT
    = "de.cau.cs.kieler.kiml.grana.edgeCount";

/**
 * Identifier for "edge direction analysis".
 */
private static final String GRANA_EDGE_DIRECTION_COUNT
    = "de.cau.cs.kieler.kiml.grana.edgeDirections";

public class UpwardnessMetric implements IAnalysis {

    /**
     * Returns a Float object that indicates the degree of 'upwardness'
     * in the given KGraph within the range of (0.0, 1.0), where 0.0 means
     * no upwardness and 1.0 means maximum upwardness.
     * This metric is based on the edge count and on the edge
     * direction analysis.
     */
    public Object doAnalysis(
            final KNode parentNode,
            final Map<String, Object> results,
            final IKielerProgressMonitor progressMonitor)
            throws KielerException {

        progressMonitor.begin("Upwardness metric analysis", 1);

        Float result;

        try {
            // load numbers from analyses
            Object edgesResult = results.get(GRANA_EDGE_COUNT);
            int totalEdgesCount = (Integer) edgesResult;

            Object[] edgeDirectionResult
                = (Object[]) results.get(GRANA_EDGE_DIRECTION_COUNT);
            int upwardEdgesCount = (Integer) edgeDirectionResult[0];

            if (totalEdgesCount > 0) {
                result = (float) upwardEdgesCount / totalEdgesCount;
            } else {
                result = 1.0f;
            }

        } finally {
            // We must close the monitor.
            progressMonitor.done();
        }

        return result;
    }
}
```

method named `doAnalysis`. This method is called by GrAna when the analysis is requested. The parameter `parentNode` specifies the parent node of the layout graph to be processed. By `results`, a map of results from auxiliary analyses may be passed. GrAna provides for defining analyses that base on the result of other analyses for reasons of efficiency. The requested analyses are scheduled in a way that respects the respective dependencies.

As many metrics relate to basic features such as the number of nodes, it is more efficient to base those metrics on auxiliary analyses which are performed first. In this case, the edge count analysis and the edge directions analysis are used as auxiliary analyses.

A progress monitor is passed by `progressMonitor`. The implementation of `doAnalysis` is expected to report progress to the monitor.

Listing 3.2: Section of `plugin.xml`

```
<extension point="de.cau.cs.kieler.kiml.grana.analysisProviders">
  <provider
    category="de.cau.cs.kieler.kiml.evol.metricCategory"
    class="de.cau.cs.kieler.kiml.evol.metrics.UpwardnessMetric"
    description="Measures the fraction of upward edges."
    id="de.cau.cs.kieler.kiml.evol.upwardnessMetric"
    name="Upwardness Metric">
    <dependency
      analysis="de.cau.cs.kieler.kiml.grana.edgeCount">
    </dependency>
    <dependency
      analysis="de.cau.cs.kieler.kiml.grana.edgeDirections">
    </dependency>
  </provider>
  <provider>...</provider>
  ...
  <provider>...</provider>
</extension>
```

In order to use this metric, EVOL needs to provide GrAna some information about it. The respective section of the `plugin.xml` for EVOL which makes the metric accessible for GrAna looks as is shown in Listing 3.3. Note that the category reads `....evol.metricCategory`, as required by EVOL. Moreover, note that two dependencies are stated here, which indicate the use of the two auxiliary analyses.

3.4 Multi-Diagram Mode

Despite the problems that are associated with the multi-diagram evaluation and that are discussed in Section 2.5, a multi-diagram mode is implemented in EVOL.

When this multi-diagram mode is switched on, each individual is successively applied to all the diagrams in the open editor windows, *i. e.* individually calculated layouts are imposed on the respective diagrams.

4 User Interface

This chapter describes the EVOL UI, which is integrated into the Eclipse GUI.

The main component of the EVOL UI is the Evolution view. It consists of a *table viewer* and a *tool bar*. A screenshot of the Evolution view in combination with other KIELER components is shown in Figure 4.1. The following sections describe the elements of the Evolution view in particular.

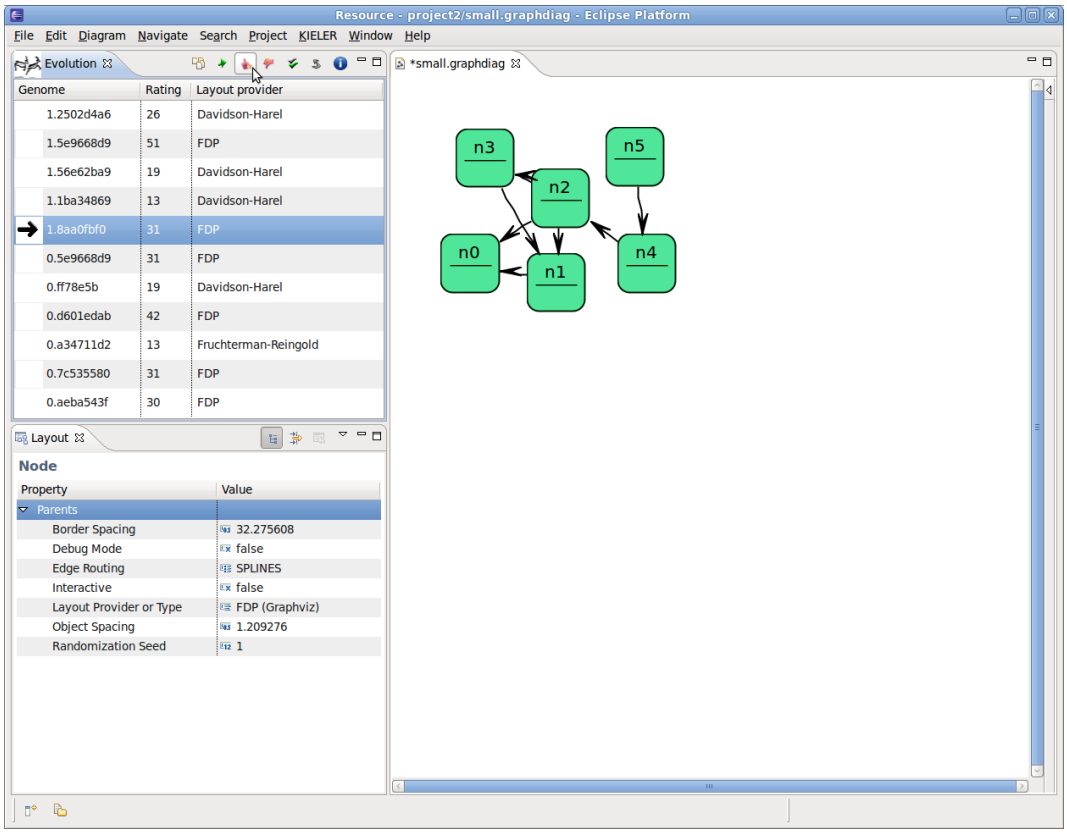


Figure 4.1: Screenshot of the Evolution view (top left) in combination with the KIML Layout view (bottom left) and a diagram editor (right) in KIELER

4.1 Population Table Viewer

The largest component of the Evolution view is the population table viewer. This table viewer shows the individuals of the current population, one per row. It has three columns. The first column contains the identifier of the individual. This allows one to identify an individual after its position in the table has changed. The identifier is an alphanumeric string which consists of the number of the generation the individual belongs to, followed by a period (.) and some hash code that has been calculated for the individual. The hash code serves the purpose of making the identifier of each individual practically unique. The generation number is added so that a distinction can be drawn between old and new individuals. For example, `4.e23a2ecd` might be such an identifier. The individual carrying that name belongs to the fourth generation, hence the initial `4`.

The second column shows the percentaged rating of the individual, rounded to an integer. A higher number means a better rating.

The third column shows the name of the layout algorithm that is encoded in the individual.

One individual can be selected at a time. This individual is referred to as the *current individual*.

Clicking on an individual in the population table view triggers the application of the respective layout on the diagram in the editor or editors.

4.2 Tool Bar

At the top of the Evolution view is the tool bar. It provides some command buttons that can be used to control the evolutionary algorithm and to inspect the candidate solutions that are generated.

Single/Multiple Editors By clicking this toggle button, the user may switch between *current editor mode* and *all editors mode*. In *current editor mode*, layout is calculated only for the diagram in the currently active editor, whereas it is applied to all visible diagram editors in *all editors mode*.

Favor By clicking Favor, the user can increase the rating of the current individual in order to promote it.

Disregard The Disregard button serves to demote the current individual. Its rating is decreased when the user clicks this button.

Evolve The most important button is Evolve. Clicking Evolve performs a step of the evolutionary algorithm. This produces a new population, based on the current population and the given ratings, that is subsequently displayed in the population table view. The first of the new individuals becomes the new current individual.

Reset The evolutionary model can be reset by clicking Reset. This makes the evolutionary algorithm re-initialize. The current population is discarded entirely and a new one is created.

Auto-rate All As the name suggests, the Auto-rate All button serves to re-calculate an automatic rating for each individual of the current population. This refreshes all the automatic ratings in the view.

Info Clicking the Info button provides detailed information about the population. This can be used by experienced users, *e. g.* for debugging purposes.

4.3 Preference Page

A rather hidden part of the EVOL UI is its preference page. In Eclipse, preference pages serve as a means of setting user preferences for plug-ins. The preference page can be accessed via the Eclipse menu bar. Select *Window→Preferences* to open the preferences dialog. In this dialog, select *KIELER* and then *KIML Evolutionary* from the list. This brings out the preference page for EVOL. The following preferences may be set for EVOL.

Population size The population size is the number of individuals that are created initially when the EA is started. This should be a small positive integer value. Its maximum value is defined in the plug-in.

Set layout provider This option indicates whether EVOL is allowed to set the layout provider when applying an individual to an editor.

Further options could be added to EVOL by adding appropriate field editors to `EvolPreferencePage.java`. Default values for these options can be set in `EvolPreferenceInitializer.java`. The preference constants that identify the options should be defined in `EvolPlugin.java`. Say for example we want to add an option for the mutation application probability. Then we would add a preference constant for it in `EvolPlugin.java`, say `mutationApplicationProbability`, which serves as identifier. In `EvolPreferenceInitializer`, we find the method `initializeDefaultPreferences()`. As the name suggests, it initializes the preferences with default values. Here we would insert a line to write the initial value for the mutation application probability into the preference store. To make the new option visible in the preference page, a corresponding field editor needs to be added in `EvolPreferencePage.java`. An appropriate choice for this option might be a `FloatFieldEditor`, as found in the KIELER UI package.

5 Conclusions

This thesis has presented EVOL, an approach to meta layout that uses EAs. EVOL contributes to the KIELER project as an add-on to KIML.

EVOL contains an EA framework that is used to evolve layout configurations, a UI to collect user feedback on created layouts, and an automatic rating prediction system which supports the user by providing rating proposals based on layout metrics.

5.1 Potential Applications

Two main applications of the presented approach can be thought of. One potential application of EVOL could be as a support for users who use automatic layout, but are not satisfied with the results produced with the default settings. This means that they need the automatic layout to be configured individually. EVOL can help those users in the configuration of automatic layout. Using EVOL, there is no need to understand what the layout options mean and how they should be set. It is not necessary for a user to know which layout options are available. It is not even necessary to know anything about layout options. Users can rely on EVOL to find some suitable settings that create "nice" layout results according to their likings.

Another potential application could be to use EVOL in the development of layout algorithms. EVOL can support developers of layout algorithms by finding suitable default values for the parameters that control the layout process. Moreover, EVOL can be used as an aid in the evaluation of layout algorithms, especially for testing robustness. As various random combinations of layout settings are tested, the robustness of layout algorithms is put to the test—also for combinations of settings the developer may not have thought of. Thus, hidden flaws in the algorithm may be revealed. Furthermore, this approach can be used to test the robustness toward invalid settings. For this purpose, the developer would permit the settings to take invalid values, and then see how the algorithm copes with one or more invalid settings. Moreover, EVOL may suggest how competitive a newly developed algorithm is in comparison to others. If an algorithm produces competitive results, these will likely endure the evolutionary process. Therefore, survival of layout configurations that contain the choice for a certain algorithm suggests that this algorithm is competitive. However, future work could enhance this use case by introducing advanced methods for comparison and evaluation, *e. g.* simultaneous evaluation of multiple individuals.

5.2 Future Work

In the course of this project it has been established that the field of meta layout provides a plethora of possibilities for future work.

This section presents some of the various ideas that arise from the work with EVOL.

Randomized Layout Algorithms Need to Be Tamed As stated in Chapter 2, randomized layout algorithms without a controllable random seed number s are a problem for optimizers, because their resulting layouts may be different in every run, and therefore, the resulting fitness values are not reproducible. It is possible that an exceptional layout is produced and rated. This problem could be mitigated by running the layout algorithm repeatedly with the same options, analyzing the resulting layouts and computing the average fitness of the several runs, thus hopefully obtaining fitness values that are sufficiently independent of the concrete settings of s. The user would also need to rate a number of layouts instead of only one. The drawbacks of this approach would be the loss of efficiency caused by the additional amount of running time that is needed for the extra runs, and the extra work the user would have to do. Apart from that, we recommend the developers of randomized layout algorithms to set rigorous limits to the random behavior in order to keep things controllable.

Rating Prediction Needs to Be Improved While the concept of rating predictors appears to be a helpful means which already produces results of passable quality, it still has a lot of improvement opportunities. Firstly, the author presumes that there is room for enhancing the accuracy and convergence speed of the rating prediction. Considering results from MOEA research could help in that task. Moreover, the rating prediction should be made more transparent. So far it takes place behind curtains. The user may control it only indirectly and has no insight in its "magical" interior. For a novice user, this is presumably the desired behavior, in that it keeps the UI simple, but advanced users might like to inspect and adjust the rating prediction. Moreover, one could think of persisting the learnt priorities of the metrics, in order to use them in subsequent runs or make them available to other users. Furthermore, it would be nice if one could automatically identify which layout options have an impact on which features of the drawing. One could then selectively optimize these options toward a defined goal, like "Maximize symmetry" or "Minimize used area", or whatever metric is preferred by the user. It might be worth considering to make use of Artificial Neural Networks (ANNs) for that purpose.

More Metrics Are Needed The set of metrics that is shipped with EVOL contains metrics for some of the most important aesthetics mentioned in literature, such as "minimize edge crossings" and "minimize edge bends", see Purchase *et al.* [25]. However, this set should be extended, in order to cover more use cases and special preferences. Among the interesting features that may influence the perception of a

drawing which are not covered yet are the angles between adjacent incident edges, and inherent symmetry. Purchase *et al.* propose both a metric for maximizing the minimum angle between edges leaving a node, and a symmetry metric. However, they state that the latter one is very computationally expensive.

Another interesting metric that has not been implemented yet is that of white-space percentage. This would give the ratio of how much of the area of the bounding box around the diagram is not occupied by the nodes' drawing.

Concerning the handling of metrics, it should be possible to configure which metrics EVOL should use. Thus computationally expensive metrics could be switched off by default. The user could be given the possibility to switch them on, *e. g.* by an option on the preference page.

Regard the Structure of Input Graphs By the use of aesthetics metrics, observable features in the appearance of graph drawings are measured. The automatic rating mechanism implemented in EVOL learns how important the user finds which aesthetic. This way, the output of layout algorithms is classified according to the weighted metrics.

However, it might be interesting to find out how the structure and type of the input graphs influence the user's preferences. For example, one may assume that a user might want a different layout for a graph with many nodes, as opposed to a small graph. In the same way, the user's preference might be dependent of density, planarity, number of self-loops, hierarchy depth, *etc.*

Exploiting *a priori* knowledge about the graph structure could possibly aid in finding a better initial population for the evolutionary process. Moreover, it is known that some layout algorithms are not designed for general graphs, but for certain types of graphs; *e. g.* an algorithm for tree layout is not apt for general graphs, except for trees. Depending on its robustness, it would produce "miserable" layout results or it might even crash if it is applied to a non-tree graph. Thus, if one knows in advance that a given graph does not fulfill the requirements for a certain layout algorithm, then it is clear that this algorithm would not be able to produce appealing output at all. The algorithm should therefore be excluded from the evolutionary process automatically. Since the GrAna framework provides not only drawing analyses, but also graph analyses, it could be used to obtain the necessary knowledge about the graph structure.

Support For Hierarchical and Partial Options Should Be Added to Evolution While KIML supports hierarchical layout, which means that layout options may be assigned to child parts of a hierarchical diagram, EVOL only supports options that apply to the whole diagram. It is conceivable that users' preferences may vary with the level of hierarchy. Furthermore, layout options should be considered that are set individually for some or all nodes, edges, or ports instead of the whole diagram. An approach to this could possibly benefit from the introduction of tree-like individuals, *i. e.* individuals with an internal structure that reflects the hierarchy of layout options.

Special properties of particular diagram types should be taken into account. For example Syncharts always have a particular top-level node.

General Persistency Concept for Evolution Configurations More basic work needs to be done on a more general concept for loading and saving of evolution configurations. It is to be found out what kind of information one should store and where to store it, *e. g.* in some type of file or in a database. One needs to consider how this is to be integrated into KIML.

Provide More Genetic Information in the Population Table The Population View could be improved so that it shows more details about the individuals, especially the values encoded in the genes. This could be achieved by adding more columns to the view. The user should be put into a position to hide or show these additional columns selectively. Alternatively the population could be displayed in a tree viewer, which may also have columns. A tree viewer would have the advantage that it could also be used to display tree-like individuals.

Furthermore, an additional view could be provided that is linked with the Population View and shows even more details about the current individual in a *one property per row* fashion.

Allow More User Intervention in Evolution The presented implementation of EVOL allows the user to influence the evolution of layout configurations to some extent by iterating stepwise through the evolutionary algorithm and rating the presented proposals. For an advanced user it would be desirable to have more opportunities to intervene, for example being able to adjust the variance for a layout option, or to select two or more interesting individuals for recombination explicitly, or to compose entire populations from interesting examples.

Evaluate Multiple Individuals Simultaneously The number of individuals the user may evaluate at a time is limited to one. Although by multi-diagram mode it is possible to view the current individuals applied to multiple diagrams, each in its own editor window, this is still limited to one individual. In order to compare two individuals on the same graph, the user has to switch back and forth between the individuals. To overcome this limitation, an approach is needed to present several diagrams of the same model side by side in the same viewer or in adjacent viewers. Each diagram would have to be zoomed to a small level. This would not only facilitate comparisons between individuals, but one could also think of a rating mode where the user is shown several individuals at a time and is asked just to pick the one he or she prefers.

Further Integration into KIELER Currently, the EVOL UI brings its own tool bar which contains controls that are defined for EVOL only. To make EVOL adhere more to the KIELER "Look and Feel", further work could investigate if the KIELER

Execution Manager (KIEM) [1] could be used as a controller for the evolutionary model. Although KIEM is primarily intended to step through the simulation of graphical domain specific models, it could possibly allow one to step generation-wise through the populations that are produced by the EA. If populations of previous generations were stored, one could also move backwards. An obstacle might be that in KIEM there is no equivalent to the rating buttons. Thus, the controls for the evaluation of individuals would still have to be provided by the Evolution view.

[1] http://rtsys.informatik.uni-kiel.de/trac/kieler/wiki/Projects/KIEM

Bibliography

[1] P.J. Angeline. Adaptive and self-adaptive evolutionary computations. In *Computational Intelligence: A Dynamic Systems Perspective*. Citeseer, 1995.

[2] C. Bennett, J. Ryall, L. Spalteholz, and A. Gooch. The aesthetics of graph visualization. *Computational Aesthetics in Graphics, Visualization, and Imaging*, 2007.

[3] T. Biedl, J. Marks, K. Ryall, and S. Whitesides. Graph multidrawing: Finding nice drawings without defining nice. In *Graph Drawing*, pages 347–355. Springer, 1998.

[4] T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.

[5] C.A. Coello Coello. A short tutorial on evolutionary multiobjective optimization. In *Evolutionary Multi-Criterion Optimization*, pages 21–40. Springer, 2001.

[6] M.K. Coleman and D.S. Parker. Aesthetics-based graph layout for human consumption. *Software: Practice and Experience*, 26(12):1415–1438, 1996.

[7] H. de Garis. Genetic programming. In *Machine learning: proceedings of the seventh international conference (1990), University of Texas, Austin, Texas, June 21-23, 1990*, page 132. Morgan Kaufmann, 1990.

[8] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry-Theory and Application*, 4(5):235–282, 1994.

[9] M. Dianati, I. Song, and M. Treiber. An introduction to genetic algorithms and evolution strategies. *University of Waterloo, Canada*, 2002.

[10] H.A.D. do Nascimento, P. Eades, and University of Sydney. School of Information Technologies. *A focus and constraint-based genetic algorithm for interactive directed graph drawing*. Citeseer, 2002.

[11] H. Fuhrmann and R. von Hanxleden. On the pragmatics of model-based design. *Foundations of Computer Software. Future Trends and Techniques for Development*, pages 116–140, 2010.

[12] H. Fuhrmann and R. von Hanxleden. Taming graphical modeling. *Model Driven Engineering Languages and Systems*, pages 196–210, 2010.

[13] C. Fujiko and J. Dickinson. Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 236–240. L. Erlbaum Associates Inc., 1987.

[14] A. Ghosh and S. Dehuri. Evolutionary algorithms for multi-criterion optimization: A survey. *International Journal of Computing & Information Sciences*, 2(1):38–57, 2004.

[15] L. J. Groves, Z. Michalewicz, P. V. Elia, and C. Z. Janikow. Genetic algorithms for drawing directed graphs. In *Proceedings of the Fifth International Symposium on Methodologies for Intelligent Systems*, pages 268–276. Elsevier Science Ltd, 1990.

[16] J.W. Gudenberg, A.N.M. Ebner, and H. Eichelberger. An Evolutionary Algorithm for the Layout of UML Class Diagrams. 2006.

[17] R. Kamalian, E. Yeh, Y. Zhang, A.M. Agogino, and H. Takagi. Reducing human fatigue in interactive evolutionary computation through fuzzy systems and machine learning systems. In *2006 IEEE International Conference on Fuzzy Systems*, pages 678–684, 2006.

[18] R. Kamalian, Y. Zhang, H. Takagi, and A.M. Agogino. Reduced human fatigue interactive evolutionary computation for micromachine design. In *Proceedings of 2005 International Conference on Machine Learning and Cybernetics, 2005*, pages 5666–5671, 2005.

[19] J.R. Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University Stanford, CA, USA, 1990.

[20] Xavier Llorà, Kumara Sastry, David E. Goldberg, Abhimanyu Gupta, and Lalitha Lakshmi. Combating user fatigue in iGAs: partial ordering, support vector machines, and synthetic fitness. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1363–1370, New York, NY, USA, 2005. ACM.

[21] T. Masui. Evolutionary learning of graph layout constraints from examples. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, page 108. ACM, 1994.

[22] D. Pallez, P. Collard, T. Baccino, and L. Dumercy. Eye-Tracking Evolutionary Algorithm to minimize user fatigue in IEC applied to Interactive One-Max problem. In *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 2883–2886. ACM, 2007.

[23] R. Poli, W.B. Langdon, N.F. McPhee, and J.R. Koza. Genetic programming: An introductory tutorial and a survey of techniques and applications. *University of Essex, UK, Tech. Rep. CES-475*, 2007.

[24] H. Purchase, R. Cohen, and M. James. Validating graph drawing aesthetics. In *Graph Drawing*, pages 435–446. Springer, 1996.

[25] H.C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages & Computing*, 13(5):501–516, 2002.

[26] Martin Rieß. A Graph Editor for Algorithm Engineering. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mri-bt.pdf`.

[27] A. Rosete-Suarez, M. Sebag, and A. Ochoa-Rodriguez. A study of evolutionary graph drawing. Technical report, Universite Paris-Sud XI, 1999.

[28] Y. Sáez, P. Isasi, J. Segovia, and J.C. Hernández. Reference chromosome to overcome user fatigue in IEC. *New Generation Computing*, 23(2):129–142, 2005.

[29] Y. Semet. Interactive Evolutionary Computation: a survey of existing theory. *University of Illinois*, 2002.

[30] W. Spears, K. De Jong, T. Bäck, D. Fogel, and H. De Garis. An overview of evolutionary computation. In *Machine Learning: ECML-93*, pages 442–459. Springer, 1993.

[31] H. Takagi. Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. *Proceedings of the IEEE*, 89(9):1275–1296, 2002.

[32] A.G.B. Tettamanzi. Drawing graphs with evolutionary algorithms. In *Proceedings of 1998 Conference on Adaptive Computing in Design and Manufacture, ACDM'98*. Citeseer, 1998.

[33] J. Utech, J. Branke, H. Schmeck, and P. Eades. An evolutionary algorithm for drawing directed graphs. In *Proc. of the Int. Conf. on Imaging Science, Systems and Technology*, pages 154–160. Citeseer, 1998.

[34] D.A.V. Veldhuizen and G.B. Lamont. Multiobjective evolutionary algorithms: Analyzing the state-of-the-art. *Evolutionary computation*, 8(2):125–147, 2000.

[35] Jonas Völcker. A quantitative analysis of Statechart aesthetics and Statechart development methods. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jovo-dt.pdf`.

[36] D. Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and software technology*, 43(14):817–831, 2001.