Evolutionary Layout of Graph Transformation Sequences

Susanne Jucknath-John and Dennis Graf and Gabriele Taentzer

12 pages, 2006

Electronic Communications of the EASST

Evolutionary Layout of Graph Transformation Sequences

Susanne Jucknath-John * and Dennis Graf * and Gabriele Taentzer *
*Technical University of Berlin, Germany

Abstract.   *Graph transformation is used in various different research areas and has been implemented in several tool environments. However, the layout of graph transformation sequences is often perceived as not optimal and remains to be a difficult task. This is partly due to the slightly different requirements for layouting graph transformation sequences compared to standard graph sequences. In this paper, we clearly define these special requirements and present a layout algorithm which fulfills them. This layout algorithm allows the user to keep track of changes during transformation steps by introducing a concept of node aging and protection of senior node positions in the layout. Furthermore, this layout algorithm introduces a concept of layout patterns. We extended the well-known spring embedder layout algorithm by these new concepts and implemented the new algorithm in AGG, an environment for Attributed Graph Grammars. The layout algorithm has been tested with various graph grammars. A brief outlook describes how this layout algorithm can also be used for different kinds of graph sequences, e.g. sequences of successively developing class diagrams.*

Keywords: graphs, graph layout, graph transformation

1   Introduction

The motivation for this work arose in two different areas. We started with the problem to preserve the mental map [1] of sucessively developing class diagrams. Available IDEs like Eclipse with an UML plugin by Omondo usually layout each class diagram separately which makes it hard to realize the areas of the graph which have changed, vanished or were newly introduced. Our approach here is to consider sequences of class diagrams not sequences of single diagrams, i.e. to consider the evolution of one initial diagram over time. Such an evolution can be naturally described by a graph transformation sequence. The next problem arose to layout graph transformation sequences adequately. A problem which also has not been sufficiently handled by the available tools. Usually graph transformation sequences can be considered as incomplete graph sequences in the sense that new graphs can be added as long as rules are applicable. Thus, special layout problems arise also here. Since graph transformation is a research area by its own with a wide range of applications, we focus on layouting of graph transformation sequences in this paper and present an implementation of a new layout algorithm in AGG [2] (an environment for

Attributed Graph Grammars).

The common requirements for layouting a graph sequence include the highest single layout quality and the lowest difference between two successive layouts. We extended this set of requirements by taking also the layout of transformation rule into account and by the option for future extensions of graph sequences without ruining the layout. Existing layout algorithms fulfill the common requirements, but not the additional requirements for layouting graph transformation sequences. Therefore, we extended an existing layout algorithm, the well-known spring embedder [6], by two new concepts: First, the concept of node aging during the sequence with the protection of senior nodes, and second, the concept of layout patterns have been incorporated. The persistence of node positions throughout a sequence is most helpful to preserve the mental map. Especially the concept of aging was the reason to name our algorithm an evolutionary layout algorithm. The second concept presents layout patterns as a visible representation of a transformation rule.

The paper is organized as follows: In Section 2, we give a short overview on an integrated development environment for graph transformation system, called AGG, and review the main graph drawing concepts as far as needed to present our new layout algorithm. The evolutionary layout algorithm itself is explained in Section 3. The implementation of this algorithm as well as its evaluation along two sample graph grammars are presented in Section 4. The conclusion and outlook follow in Section 5.

2 Background

First, we give a short overview on the graph transformation environment AGG which function as platform for the implementation of our new layout algorithm. Thereafter, we focus on the necessary background for presenting the new layout algorithm, including the offline graph drawing problem, layout quality metrics and the original spring-embedder layout.

2.1 Graph Transformation Environments

Several graph transformation environments, like Progres [2], AGG [3], Fujaba [4], Groove [5], have been developed with diverse functionality offered. Although these environments differ in the kinds of graphs and their manipulation, they all have to present graphs resulting from transformations. Therefore they meet similar problems in layouting graph transformation sequences. We have used AGG as a case study on requirements for a layout on graph transformation sequences and integrated our resulting layout algorithm into this tool.

AGG [3] is an integrated development environment for algebraic graph transformation systems. It consists of several visual editors to develop graphs and rules, an interpreter, and a set of validation tools for graph transformation systems. The editor provides a visual layout of AGG graphs similar to UML object diagrams. For a first impression of AGG see the screen shot in Fig. 1. The visual interpreter supports transformation of graphs in two different modes: stepwise and as long as possible. AGG supports the algebraic approach to graph transformation [7]. Its graphs may be typed and attributed by Java objects. Note that in AGG, the positions of nodes and edges do not store any syntactic or semantic information, i.e., the layout of a graph is just a matter of presentation for the sake of readability. But the layout is of considerable importance for a user, although it does not bear any relevant information in the
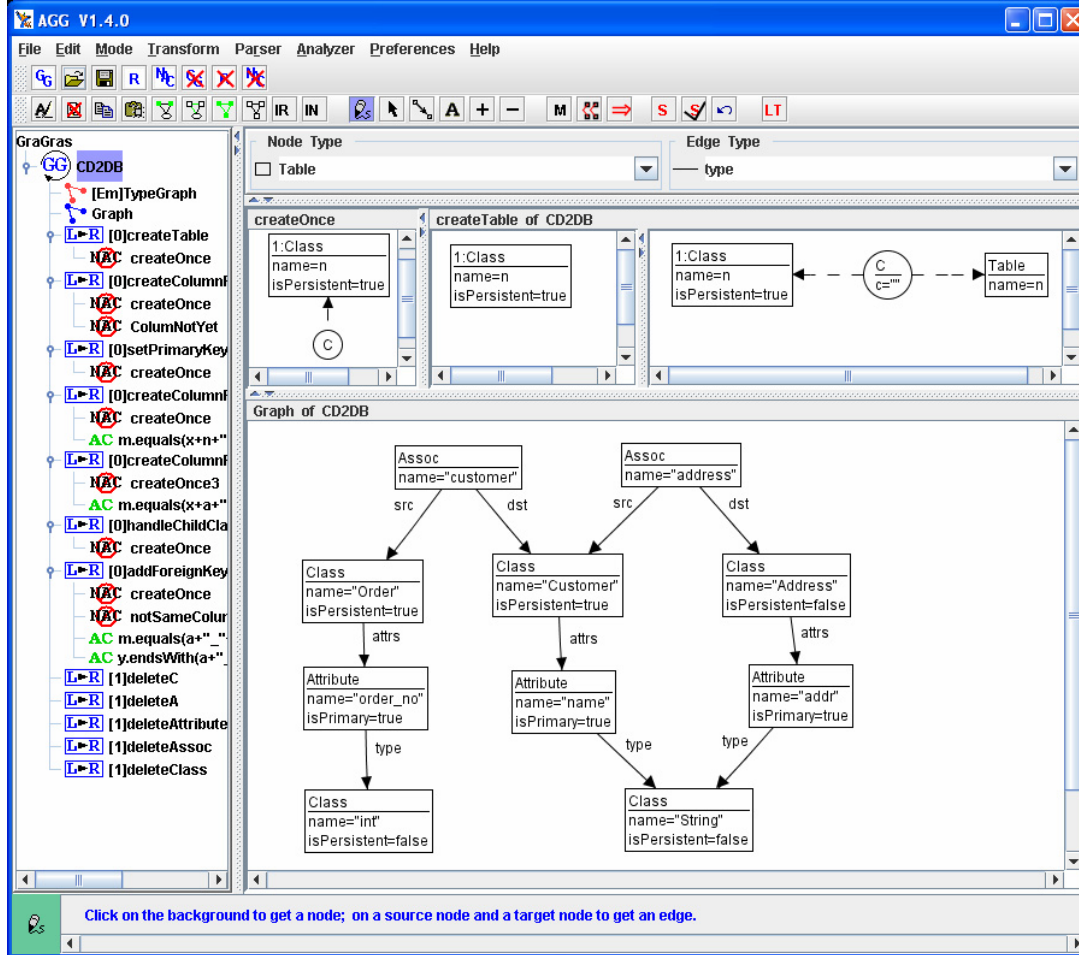
Figure 1: AGG - Attributed Graph Grammar Environment

first place. Due to the fact that the problem of automatically computing a reasonable layout of graph transformation sequences is hard, it has been solved in AGG in a quite simple way up to now.

2.2 Offline Graph Drawing Problem

The offline graph drawing problem [8] describes the following trade-off for a given graph sequence $G_0...G_n$ and their corresponding layouts $L_0...L_n$:

- The quality ρ_{L_i} of every single layout L_i should be as optimal as possible.

- The mental distance $\delta_{L_i,L_{i+1}}$ from one layout L_i to its successor layout L_{i+1} should be small.

- The resulting quality Q of the whole sequence results in the sum of every ρ_{L_i} minus every $\delta_{L_i,L_{i+1}}$,

$$Q_{L_0..L_n} = \sum_{i=0}^{n} \rho_{L_i} - \sum_{i=1}^{n} \delta_{L_{i-1},L_i} \tag{1}$$

Approaches to measure ρ (the single graph layout quality) and δ (the mental distance between two graphs) have been described in several articles beforehand [9, 10, 11, 12, 13]. We present and discuss a specific definition for ρ and δ in our example in Section 4.

The offline graph drawing problem has been solved for complete sequences of graphs by foresighted graph layout [14]. Foresighted graph layout offers a number of different approaches as e.g. to use a super graph which includes every graph of the sequence as subgraph. Thus the layout of every subgraph is part of the super graph layout. The position of a single node is very steady this way and the chance to find a compromise between the requirements for single layouts is rather good. But this compromise layout works only as long as no new graphs are added to the sequence. This is the reason, why this solution does not fit well for incomplete graph sequences.

2.3 Spring Embedder Layout

The spring embedder layout[6] is a force-driven layout which is based on an energy model. Every node in this energy model contains attractive forces f_a and repulsive forces f_r. Therefore, the edges are modeled as springs which have a preferred length, but which may be stretched or compressed to serve the node's attracting or repulsive forces. Attracting forces occur between incident nodes, while nodes which are not connected by edges, are separated by repulsive forces. The attracting forces f_a are described by

$$f_a(d) = \frac{d^2}{k} \tag{2}$$

and the repulsive forces f_r are described by

$$f_r(d) = -\frac{k^2}{d} \tag{3}$$

where d describes the current distance of two nodes and k denotes the minimal space around a node which should be empty. The algorithm computes the layout in an iterative way. In each iteration step, the sum of all attracting and all repulsive forces is computed for each node. Then, each node gets a new position such that the overall energy is reduced. The aim is a system which is as balanced as possible, i.e. each node has as little energy as possible. The next idea is to give nodes a higher flexibility to move around in first iterations and to restrict this general flexibility with each further iteration until the nodes reach a stage where they can barely move. This process can be compared with a cooling down process, thus the function to describe this process is called the *cooling* function. That means nodes is given a temperature which defines a maximum velocity at each iteration step.

3 Evolutionary Layout of a Graph Transformations Sequence

This section includes the problem definition, our new layout algorithm and its implementation.

3.1 Problem definition

The requirements for layouting instance graphs can be described by the offline graph drawing problem as discussed in the previous section. We extend this problem description by additional requirements for incomplete graph sequences: For a given graph sequence $G_0...G_n$ with their corresponding layouts $L_0..L_n$:

1. The quality ρ_{L_i} of every single layout L_i should be as optimal as possible.

2. The mental distance $\delta_{L_i,L_{i+1}}$ from one layout L_i to its successor layout L_{i+1} should be small.

3. The graph sequence may have future extensions, without losing the layout's quality.

4. The changes between two graphs should be easily recognizable in the layout of two subsequent graphs, e.g. the layout of the corresponding transformation rule should be taken into account.

The original graph drawing problem covers only the first and the second topic.

3.2 Evolutionary Layout Algorithm

None of the existing graph layout algorithms we have considered beforehand, fulfilled all the requirements given above. This is not to say that they are doomed to produce a non-optimal layout, but it depends on the specific data how good the layout turns out to be. Especially changes by transformation rules can or cannot be recognizable in the layout, dependent on if they are part of the algorithm specification. We wanted to improve the situation by introducing these requirements to a layout algorithm and decided to adapt an existing algorithm. A spring-embedder layout [6] is easily extensible, since its energy model suits very well for integration new requirements. Even if the requirements are rivals (best single layout vs. least mental distance between two layouts), each requirement can get a certain amount of impact on the whole layout. A further requirement concerns the specification of graphs based on rules. To support the user's recognition of changes by a transformation rule in the instance view, we relate each transformation rule to a specific layout and call this combination a layout pattern.

Iterative Layout Computation   The first graph of a sequence is taken as an initial graph and its layout is not to be altered by other layouts. Usually, the layout of graph transformation sequences does not distinguish between different node types, thus attracting forces f_a are the same for all nodes and repulsive forces f_r exist as soon as two nodes are placed nearby. Thus, attracting and repulsive forces are computed as in the spring embedder layout.

The original spring embedder layout algorithm uses the metaphor of a temperature to express the position flexibility of each node and a cooling function to reduce this temperature in each iteration of the algorithm. This asymptotic cooling function of the temperature t is described by

$$t(i) = t(i-1) - \frac{t(i-1)}{i+1} \tag{4}$$

for the iterations with $i > 0$, and by an initial temperature $t(0) = 100$ for the initial layout. An alternative asymptotic function could also be used to describe the cooling process.

Protection of senior nodes   The mental difference between two layouts can be reduced, if the same node has similar positions in both layouts. To fulfill this requirement, we introduced the concept of aging where a node gets older if it is included in the next graph. The older a node is, the less it should be moved. If possible, a younger node is moved instead. This can be realized by extending the cooling function by a specific temperature $t_N(i)$ for each node N in iteration step i by integrating its age a_N:

$$t_N(i) = t_N(i-1) - \frac{t_N(i-1)}{i - a_N + 1} \tag{5}$$

This way, the older nodes have less movement flexibility. Since the nodes contained in the initially layouted graph, are the eldest ones, they hardly do not have any move flexibility.

Shock-aging   The move flexibility of a single node depends highly on its age, but even more on the adjacent nodes with their attractive forces to their neighbors. If one node is taken away, its former neighbors have abruptly much more move flexibility. This can cause dramatic changes in the layout of this graph area. On the one hand, these changes draw attention which is good. On the other hand, it can ruin the viewer's mental map. Thus an extension was integrated not to stop, but to slow down the general development of occupying the space left behind by deleted nodes. This process can be realized by giving adjacent nodes some extra age, by so-called shock-aging. Of course, this can only be useful in later layouts, not in the first layout after the initial layout. The actual effect would come later, but this is acceptable, since also the mental map has to be built up in the first layouts.

Layout patterns   Besides the general limitation by the temperature, special layout patterns can limit the flexibility of individual nodes. A layout pattern is defined as a smaller graph G_{LP} with its layout L_{LP}. If G_{LP} can be recognized as a pattern in a larger graph G, then layout L of G is influenced by L_{LP}. For example, G_{LP} may be defined by a pair of parent and child nodes and L_{LP} can be defined by placing a child node always below its parent node. This layout pattern forces a graph to grow downward, and the typical appearance of a tree evolves over time.

3.3   Implementation

To integrate the new layout algorithm into AGG, the first adjustment concerns the graph structure model in AGG which originally contained very simple layout information only. For the new layout algorithm, this layout data has been enriched by the following items: node age, preferred edge length, edge-binding force (resulting from the node forces), and a kind of bounding box around each node to indicate where other nodes may not be placed.

To set transformation rules into relation with the layout of graph transformation sequences, a general analysis has to take place to determine new nodes, deleted nodes and constant nodes. The analysis results are used to define the age of nodes: the age of constant nodes is raised, while new nodes are given an age of 1. Deleted nodes are treated differently, because they change the layout quality metric and have an effect on the layout of their neighbors (shock-aging).

The implemented quality metrics are described in detail by an example in Section 4 which was generated by AGG after integrating the new layout algorithm. Other details of the integration need a deeper knowledge about the internals of AGG and thus, are not part of this paper, but they are presented in [15].

4 Examples

In this section, we evaluate the new layout algorithm by two examples. A number of quality metrics for layout are implemented in AGG and discussed for one of the examples presented.

Example: Tree Generation   The first example presents the evolution of a tree (see two graphs of the corresponding sequence in Fig. 2). The layout pattern for this graph grammar forces every child node being placed below its parent node. Furthermore, child nodes have a similar distance to each other which is caused by the spring embedder layout. This effect is not described by a pattern itself, but fits very well into the common picture of a tree with a half-radial layout.
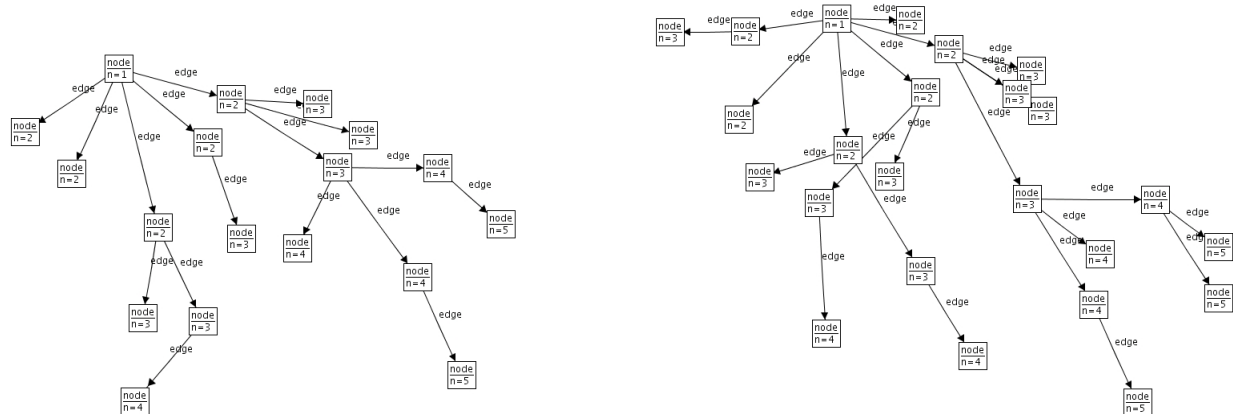

Figure 2: Tree Generation: Graphs 17 and 23

Example: Model Transformation   The second example presents the layout of a standard model transformation problem [16] which occurs in several variants. The source language consists of simple class diagrams, a sample diagram (in abstract syntax format) is presented in the main view of Fig. 1. The target language consists of schemes for database tables, one sample schema is presented in Fig. 5. A reference structure is established as helper structure for the model transformation which relates classes with tables and attributes with columns. On top of Fig. 1 a transformation rule is presented which creates a table for each top level class. Other transformation rules insert a column for each attribute and finally foreign keys (FKey) for associations.

The graph transformation sequence presented consists of 26 graphs and besides the first one (Fig. 1), we decided to present explicitly the second and third as well as the last one (Figs. 3 - 5). If the start layout is easy to understand and the following layout differences are small, then a user may keep track

even if the graph (and its layout) gets more and more complicated. To discuss this in more detail, please take a look at Table 1.

A	B	C	D	E	F	G	H	I				J	K
age	#n	#e	n_{ov}	n_{eov}	e_{xing}	e_{dif}	n_{mov}	δ_{single}				δ_{mental}	δ_{result}
0	10	10	45	0	27	9	N/A	0.2 +	20 +	0.3 =	20.5	N/A	N/A
1	12	12	0	0	1	142	0	12 +	24 +	12 =	48	11.8	36.2
2	14	15	0	3	2	82	13	14 +	7.2 +	5 =	26.2	6.4	19.8
3	14	16	0	3	2	83	8	14 +	7.5 +	5.3 =	26.8	5.8	21
4	16	18	0	2	2	81	11	16 +	11.3 +	6 =	33.3	5.2	28.1
5	19	24	0	10	6	109	80	19 +	3.9 +	3.4 =	26.3	8.7	17.6
6	22	30	0	8	6	103	75	22 +	5.8 +	4.3 =	32.1	6.8	25.3
7	24	33	0	12	16	92	100	24 +	4.4 +	1.9 =	30.3	6.9	23.4
8	25	35	0	11	15	104	185	25 +	5 +	2.2 =	32.2	10.4	21.8
9	25	36	0	11	16	101	55	25 +	5.1 +	2.1 =	32.2	5	27.2
10	24	35	0	11	15	84	77	24 +	4.9 +	2.2 =	32.1	5.6	26.5
11	23	33	0	6	15	86	76	23 +	8 +	2 =	33	6	27
12	22	31	0	15	7	84	127	22 +	3.3 +	3.9 =	29.2	8.5	20.7
13	21	27	0	5	7	52	99	21 +	8 +	3.4 =	32.4	6.6	25.8
14	20	25	0	5	6	85	116	20 +	7.5 +	3.6 =	31.1	9.2	21.9
15	19	23	0	4	4	54	115	19 +	8.4 +	4.6 =	32	8.3	23.7
16	18	21	0	1	0	28	63	18 +	19.5 +	21 =	58.5	4.8	53.7
17	17	21	0	0	0	23	44	17 +	38 +	21 =	76	3.6	72.4
18	16	19	0	3	0	36	64	16 +	11.7 +	19 =	46.7	5.9	40.8
19	15	18	0	0	0	15	104	15 +	33 +	18 =	66	7.7	58.3
20	14	17	0	1	0	45	116	14 +	15.5 +	17 =	46.5	10.9	35.6
21	13	16	0	0	0	23	115	13 +	29 +	16 =	58	10.2	47.8
22	12	15	0	2	0	29	124	12 +	9 +	15 =	36	12.2	23.8
23	11	15	0	0	0	14	183	11 +	26 +	15 =	52	17.5	34.5
24	10	14	0	0	0	12	45	10 +	24 +	14 =	48	5.3	42.7
25	9	13	0	0	0	11	86	9 +	22 +	13 =	44	10.3	33.7
26	8	12	0	0	0	12	49	8 +	20 +	12 =	40	7.1	32.9

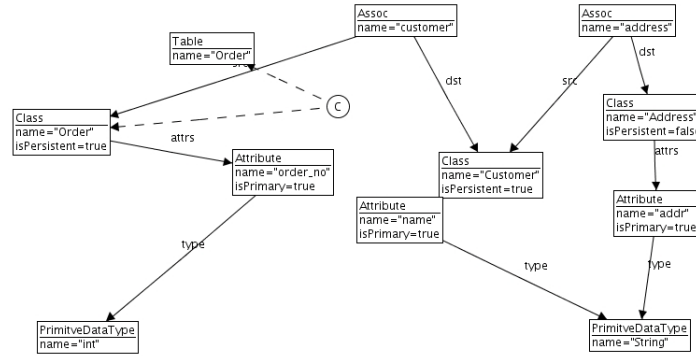Table 1: Quality Metrics for Example 2

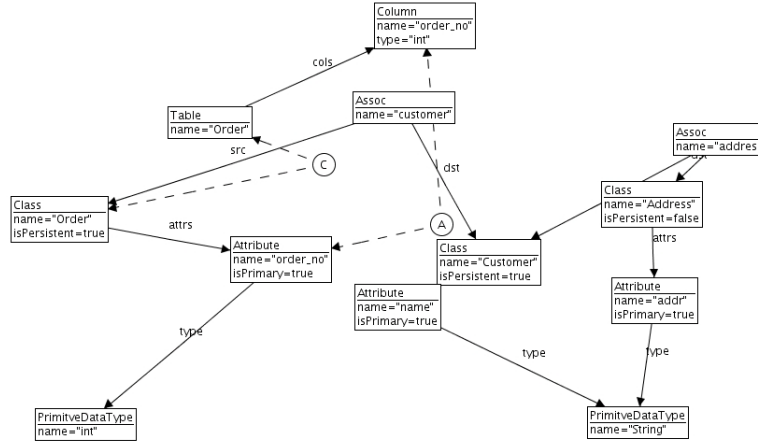Figure 3: CD2DB-Example: Graph 2
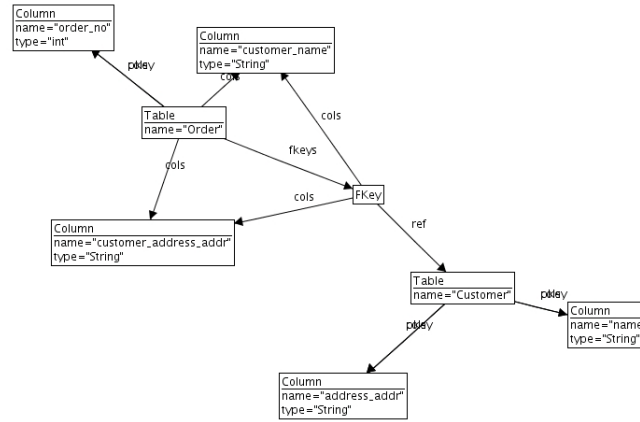
Figure 4: CD2DB-Example: Graph 3

Figure 5: CD2DB-Example: Graph 26

Columns A, B, and C describe some statistical data. Column A holds the graph position in the sequence which can be described as graph age. Columns B and C hold the number of nodes and edges, respectively. Undesired effects for a single layout are described in columns D, E, F and for the mental distance in columns G, H, the conclusions follow in columns I, J, K. The semantics of table entries is described in more detail below. These metrics are weighted according to the graph drawing problem:

1. Weighting of single layout quality δ_{single}:
   A node overlapping (D) is worse than a node-edge-overlapping (E) which is itself worse than a edge crossing (F). In an ideal layout, overlapping nodes would not occur, but they cannot always be prevented, given a large number of edges with strong coupling. Every overlapping must be seen

in relation to the number of elements:

$$\delta_{single} = \frac{B}{D+1} + \frac{(B+C)}{E+1} + \frac{C}{F+1} \tag{6}$$

2. Weighting of mental distance δ_{mental}:
   Normally, movements of nodes (H) and edges (G), both values given in pixels, are undesirable for a stable mental map. Again, a movement of elements must be seen in relation to its number:

$$\delta_{mental} = \frac{H}{B} + \frac{G}{C} \tag{7}$$

3. Resulting layout quality δ_{result}:
   To summarize these effects, we give the quality of a single layout in column I, in column J the mental distance to the previous layout and in column K the resulting quality as described by (1) are shown.

It turns out that node-overlappings do not occur after the first transformation step, a small number of node-edge overlappings do occur, and so do a larger number of edge crossings. But this is only the situation for the first 15 layouts, the next 11 layouts contain no node-overlappings, no edge-crossings and very few node-edge overlappings. It is recognizable that the general single layout quality ρ does not differ much in these first 15 layouts. This is caused by a balance of a bonus for placing more nodes without overlapping and a malus for getting more edge-crossings with more edges involved. But the increase of edge-crossings is also due to the stronger coupling in these graphs and cannot always be avoided.

First, we can summarize that the disturbance is comparatively small given the number of edges and their coupling. Second, the layout algorithm is able to turn back to a better layout in the end, given a small number of nodes. Many layout algorithms tend to stick to bad layouts, once a bad layout has been reached. This is not the case for the evolutionary layout presented. The evolutionary layout can come back to better layout, as shown e.g. in Fig. 5.

5 Conclusion

In this paper, we discussed the difficulties for layouting incomplete graph transformation sequences. We stated the requirements for a good layout of graph transformation sequences and tested several existing layout algorithms according these requirements. Although a number of solutions exist already to handle complete (e.g. non-evolving) graph sequences, all of them would have to be adapted for incomplete sequences, if possible. We have chosen to adapt the spring-embedder layout algorithm, because it suits best to serve the different requirements, even if they are contrary to each other. This is already the case if we want to get the best layout for a single graph on the one hand, and a minimal distance to the previous layout on the other hand. Our adapted algorithm can also include layout pattern to optimize future sequence expansion.

The basic idea of the new layout algorithm was to consider a graph sequence not as a series of independent graphs, but as an evolution of one graph. In this context, every node has the concept of aging, restricting the flexibility of node positions. Older nodes which exist already for several graph generations, should move less than younger nodes. One effect of this aging concept is the support of a mental map, since the viewer's mind gets used to the former graph and can easily recognize longer living parts in later successors. The conflict between the requirements, i.e. to keep a position for an older node and to have an optimal layout for the actual graph, has been expressed by an adapted spring-embedder layout. This layout algorithm implements a compromise between the best single layout for a graph and the least mental distance of a graph to its predecessor. This compromise gets more fuzzy, if additional requirements arise given by layout patterns. On the one hand, the concept of layout patterns is important to make transformation rules more recognizable in the layout. On the other hand, if a layout pattern becomes too complex, e.g. more than the one-node to two-nodes-plus-edge relation given in our example in Section 4, it may behave contrary to the user's mental map. For this reason, the usage of layout patterns is optional in our prototype implementation.

We sketched how the new layout algorithm has been integrated into AGG and discussed the value of this layout algorithm by two examples. Our algorithm produces readable layouts and can be used for a graph transformation sequence as well as for incomplete sequences of class models [17] or other graph-like diagrams which are converted to the AGG graph format. Although we motivated that existing layout algorithms are not well suited to layout incomplete graph sequences, because they do not fulfill all the requirements, it would be interesting to compare the practical results produced by different layouters for graph sequences in future work. Future work concerns improved layout patterns for class diagram sequences. Furthermore, we also like to consider developer dependency graphs [18] and to layout them adequately.

References

[1] K. Misue, P. Eades, W. Lai, and K. Sugiyama, Layout Adjustment and the Mental Map, *Journal of Visual Languages and Computing 6*, 183-210, 1995.

[2] A. Schürr, A. Winter, A. Zündorf: PROGRES: Language and Environment, in *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Application, Languages and Tools*, World Scientific, 1999.

[3] C.Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and environment, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Application, Languages and Tools*, World Scientific, 1999.

[4] T. Fischer, Jörg Niere, L. Torunski, and Albert Zündorf, Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language, in Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany (G. Engels and G. Rozenberg, eds.), LNCS 1764, pp. 296–309, Springer Verlag, November 1998. Available at http://www.fujaba.de.

[5] A. Rensink, The GROOVE simulator: A tool for state space generation. In M. Nagl and J. Pfalz, editors, Applications of Graph Transformations with Industrial Relevance (AGTIVE), Volume 3062 of Lecture Notes in Computer Science, Springer-Verlag, pp. 479-485, 2003.

[6] T.Fruchterman, E.Reingold, Graph Drawing by Force-Directed Placement, *Software Practice and Experience 21*, p. 1129 - 1164, 1991.

[7] H. Ehrig, K. Ehrig, U.Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation*, EATCS monographs, Springer, 2006

[8] S. Diehl, C.Goerg, A. Kerren, Foresighted Graphlayout, Technical Report A/02/2000, FR 6.2 - Informatik, University of Saarland, 2000.

[9] P. Eades, A Heuristic for Graph Drawing, *Congessus Numerantium 42* p. 149 - 160, 1984.

[10] S. Bridgeman, R. Tamassia, Difference Metrics for Interactive Orthogonal Graph Drawing Algorthms, *Lecture Notes in Computer Science 1547*p.57 - 71, 1998.

[11] M.K. Coleman and D.S. Parker, Aesthetics-based based Graph Layout for Human Consumption. *Software Practice and Experience*, 26(12):p.1415-1438, 1996

[12] R. Davidson and D. Harel. Drawing Graphs nicely using Simulated Annealing. *ACM Transactions on Graphics*, 15(4): p 301-331, 1996.

[13] H.C. Purchase, M. McGill, L. Colpoys and D. Carrington. Graph Drawing Aesthetics and the Comprehension of UML Class Diagrams, An empirical study, Proceedings of the Australian Symposium on Information Visualisation, Eades, P. and Pattison, T. (eds), Australian Computer society, pp129-137, 2001

[14] S. Diehl and C. Goerg, Graphs, They are Changing, *Proceedings of 10th International Symposium on Graph Drawing*, Irvine, California, August 26-28, 2002.

[15] D.Graf, *Evolutionaeres Layout von Graphsequenzen unter Nutzung von anwendungsspezifischen Vorwissen*, Master Thesis, TU Berlin, 2006

[16] *Model Transformation in Practice*, Satellite Workshop of MODELS 2005, http://sosym.dcs.kcl.ac.uk/events/mtip

[17] S. Jucknath-John, D.Graf and G. Taentzer, Preserving the Mental Map during the Development of Class Models, *Proceedings of the ACM Symposium on Software Visualization SoftVis 2006*, Brighton, UK, September 4-5, 2006.

[18] S. Jucknath-John and J.Bochnia, Code Dependencies meet Developer Dependencies, *Proceedings of IASTED International Conference on Software Engineering SE 2006*, Innsbruck, Austria, February 13-16, 2006.