On Class Diagrams, Crossings and Metrics

Holger Eichelberger

University of Würzburg
Am Hubland, 97074 Würzburg, Germany
eichelbe@informatik.uni-wuerzburg.de

Abstract. UML class diagrams, internationally specified and widely used in software engineering, are a great challenge in automatic drawing of graphs. Due to the complex nature of UML class diagrams and the requirements of software engineers, who need to read these diagrams, layout rules and algorithms for general graphs can not be applied without adaptions, extensions and modifications. In this paper, we present layout rules, a hierarchical layout algorithm and an edge crossing reduction strategy tailored for UML class diagrams. Furthermore, the problem of measuring aesthetic quality is discussed.

1 Introduction

In object oriented software engineering, class diagrams are the first choice when visualizing static structures of classes and relationships. In particular, the class diagram of the Unified Modeling Language (UML), as specified in [22], appears as a standardized and stable diagram type, which has not been changed significantly in the last years, even when establishing to the new version 2.0 of UML [23].
The automatic drawing of UML class diagrams is a great challenge for software engineering as well as software visualization: Diagrams drawn by engineers can be standardized in their visual appearance, e.g., due to cooperate company styles, data gained from reverse engineering existing programs can be laid out automatically and the visionary ideas of forward engineering by model driven architecture (MDA) transformations [21] or a round-trip cycle can be realized. Unfortunately, evaluations of current UML tools as described in [7, 11] show that most tools do not provide proper layout mechanisms and produce structurally invalid diagrams when applying the build-in layout mechanisms. Unfortunately, layout-plugins from the graph drawing community like [12, 16], but also many UML tools, tend to disregard the numerous model elements specified by the UML and, therefore, lose the compliance to the UML specification.
In this paper, we give an overview on on drawing UML class diagrams in hierarchical style. The next sections lists previous work, give a basic introduction into features of UML class diagrams and discuss our unique set of syntactical and semantical diagramming rules for UML class diagrams in conjunction with our layout algorithm realizing these rules. Then, an edge crossing reduction algorithm designed for hierarchical diagrams in introduced. Finally, aspects on measuring layout quality for UML class diagrams are discussed and conclusions on our work are listed.

2 Previous Work

In [7, 11], the automatic layout of UML class diagrams done by professional Computer Aided Software Engineering (CASE) tools has been identified as insufficient and most times invalid considering the UML specification.
From a more diagrammatic viewpoint, the automatic layout of class diagrams has been attacked by different approaches so far. On the predecessor of UML, the Object Modeling Technique (OMT), a force directed algorithm, which enforces the directions of certain types of relationships by different magnetic fields, was described in [19, 20]. Unfortunately, this work is neither maintained nor it was adapted towards UML class diagrams so far. A variant of the well-known Sugiyama algorithm [31, 32] for an early version of UML class diagrams was given in [29]. In fact, some of the basic ideas significantly influenced our work, even if adjustments to the richness of elements in more recent versions of UML and to layout quality lead to a different adaption of the Sugiyama algorithm. In [12, 16], two variants of the topology-shape-metrics method have been described. Both consider only basic elements of the UML specification and no nesting

of elements. Other approaches like [30] mainly rely on the (hierarchical) algorithms defined in some graph drawing libraries.
Much basic work for judging the quality of UML class diagrams in terms of traditional graph drawing aesthetics has been described in [24, 26, 27]. Unfortunately, the authors finally concluded that the evaluation produced no significant results or the results were difficult to interpret reasonably and consistently. Therefore, a more semantical approaches for measuring quality and aesthetics as well as layout algorithms, which consider the underlying semantics of a diagram, were requested. In [6, 8, 9], we introduced a unique set of aesthetic rules for UML class diagrams and in the latter work a framework for measuring aesthetic quality. Other approaches to layout rules like [1, 3] imposed too much restrictions or were not compliant to the UML.
On the reduction of edge crossings in usual graphs, in general a NP-complete problem, a huge set of heuristics has been proposed, e.g., the barycentric method [4, 32] or the median method [5]. On the one side, UML class diagrams may be compound graphs, because elements can be nested in other elements and such compounds may relate to each other. As suggested in [28], basic strategies can be modified to support clusters or compounds. Unfortunately, this appears not to be sufficient as shown in [13, 14]. On the other side, different types of edges in an UML class diagram lead to various types of edge crossings, which are usually not considered by heuristics designed for general or compound graphs.

3 UML Class Diagrams

At a first glance, UML class diagrams may appear as usual graphs similar to entity relationship diagrams, one of the historical predecessors of UML. From a more precise viewpoint, the variety of elements introduces a high complexity: Different types of nodes like classes, higher associations, comments or model management elements, e.g., packages, as well as different types of edges can be combined. Furthermore, hyper edges may occur at association classes, comments, UML constraints or generalizations of associations.
On the one side, it appears superfluous to realize all elements and (valid) combinations by relations, because elements like higher associations tend to appear seldom in real world diagrams. On the other side, the UML specification clearly states that compliant tools must implement all types of nodes and edges, but may select between presentation options, e.g., applying the shared target style to generalizations, aggregations or compositions, i.e., joining them at the common element. Furthermore, from our viewpoint, also a reduction of the visual complexity, e.g., by filtering elements or collapsing packages is not appropriate for a UML layout tool, because the contents of the diagram depends on the choice of the software engineer and transformations, which may change the contents of a diagram, belong to higher level mechanisms like CASE tool operations or MDA transformations.

4 Diagram Rules and Layout Algorithm

General automatic layout algorithms try to be compliant to basic diagramming rules to support readability as proposed by various publications. In [6, 8], a set of rules, designed to capture the syntax and semantics of UML class diagrams, was introduced. An exhaustive discussion on deducing these rules from the basic disciplines involved in drawing UML class diagrams, namely software engineering, software visualization, graph drawing and human computer interaction (HCI) was given in [9]. In this section, a brief summary of these rules is described. The entire set consists of 16 mandatory, 5 optional and 3 facultative user related criteria.

- Emphasize **hierarchy** due to the requirements of UML and software engineers as described in [2, 17].
- If compounds are present, the **containment** of model elements should be visualized properly. As shown in [7, 11], this rule is often not considered by commercial tools.
- Elements may be **distributed over the display area** to emphasize grouping, collaboration or coupling as introduced in [6, 8].
- A **scaling** of the elements may be appropriate depending on the semantics of the diagram. Therefore, metrics reflecting magnitude or complexity may influence the size of elements as discussed in [8, 18].

- **Overlappings** of model elements, except of those elements nested in each other, should be avoided. In particular, nodes or edges should not overlap each other, but also edges should not overlap nodes to provide readability.
- Selected nodes should appear at a **central position**, e.g., the rhombs of higher associations usually appear at a median position of the connected nodes. The same is true for highly connected comments.
- Some model elements should appear **close to connected elements**. For example, this applies to association classes, constraints or comments.
- Additionally, edge lengths, edge crossings or compactness may be considered.

Due to various conflicts between the summarized criteria listed above, e.g., compactness does not always fit to distribution or scaling of model elements, priorities on the individual criteria were introduced. The sequence of the listing above reflects relative priorities. Thereby, criteria, which are directly related to the syntax or semantics of UML class diagrams reflect a higher priority than syntactical rules.
The relative priorities were directly encoded in our drawing algorithm. In the following, a brief summary of the algorithm is given. A more detailed description with examples was presented in [9, 10]. From a coarse-grained viewpoint, it follows the well-known Sugiyama algorithm [32, 31].

- preprocessing:
  - Retrieve a **proper hierarchy**. In the implementation, certain groups of edges can be combined to specify an appropriate pseudo-hierarchy. In general, it would be amenable having an algorithm at hands, which is able to automatically retrieve a pseudo hierarchy. Unfortunately, currently no user studies on this field were published. Therefore, realizing such an algorithm would currently be guesswork only.
  - **Release input dependencies** by ordering the elements of the graph according to fully qualified names of the (connected) elements.
  - **Compress elements** which should appear in a close vicinity in the result. For example, association classes, dummy nodes representing constraints and certain comments are packed into proxy nodes in this step.
- Perform a **rank assignment** with extensions for packages. Thereby, transitive relations between elements within a compound should be considered as (invisible) direct relationships between the compounds. Furthermore, hierarchical relations to a compound as well as non-hierarchical edges in general and their connected elements must be respected.
- **Reduce the number of edge crossings**. This will be described in detail in the next section.
- **Unpack compressed elements** from proxy nodes to have the individual elements available in the coordinates assignment.
- **Calculate the coordinates** of the individual elements. First, an iterative process determines the positions of hierarchically connected elements similar to [15]. Thereby, compounds are considered. Then, edges and non-hierarchical relations are routed in a second step.
- postprocessing:
  - **Improve the positions** of association classes, constraints and comments according to the default layout style implicitly suggested by UML.
  - **Apply additional transformations** like grid positions for nodes and edges.

In contradiction to the Seemann algorithm in [29], our algorithm assigns all nodes to layers in the rank assignment step and considers all edges in the edge crossing reduction.

5 Crossing Reduction for UML Class Diagrams

According to our aesthetic rules for UML class diagrams in section 4, edges are partitioned into hierarchical and non-hierarchical edges. In our algorithm, hierarchical edges can be processed as usual in the Sugiyama algorithm and, therefore, occur at the horizontal sides of nodes. To make a clear distinction between both partitions, non-hierarchical edges are laid out in orthogonal style at the end of the coordinates assignment phase and are drawn at the vertical sides of nodes. Therefore, the following types of edge crossings may occur. Crossings between

- **hierarchical edges** can be handled according to the methods in [33], which can slightly be speeded up by an incremental calculation as described in [9].
- **hierarchical edges and non-hierarchical multi level edges**. In principle, these edges can be treated like hierarchical edges. Unfortunately, due to the orthogonal layout of non-hierarchical edges, not all edge crossings can thereby be detected. Therefore, additional dummy nodes simulating the orthogonalization to be done by the coordinates assignment later on are inserted before performing the edge crossing reduction.
- **hierarchical edges and non-hierarchical flat edges**, which connect nodes in the same layer. A matrix technique similar to that for the hierarchical edges can be used to (incrementally) calculate this number of edge crossings as shown in [9]. Thereby, also the decision is made, if a flat edge should occur at the top or the bottom side of a rank.
- **non-hierarchical flat edges**. Due to the option of assigning each flat non-hierarchical edge either to the top or the bottom side of a rank, it a constrained orthogonal layout mechanism followed by an edge crossing calculation can calculate an approximation of the optimal number of crossings crossing.
- **edges and compounds** can be considered by introducing artificial compound border nodes connected by appropriate edges, which receive a recursively decreasing edge weight (*heavy edge method*). Hence, the edge crossing number calculation should work on weighted pseudo crossing numbers.
- **edges and interior parts of nodes**, e.g., edges overlapping interior compartments of subsystems, a visually partitioned variant of packages as specified in [22]. By introducing a penalty mechanism and changing the arrangement of the inner compartments dynamically, these crossings can be avoided.

Algorithm 1 hierarchical crossing reduction

```
input: $\bar{G} = (V, E_H, E_N, n, \sigma)$ ($V$ sorted)
output: $\bar{G}^* = (V, E_H, E_N, n, \sigma^*)$
  $\bar{G}^* = (\{\}, E_H, E_N, n, \sigma^*)$
  while $|\{v \,:\, v \in V,\ isMarked(v)\}| < |V|$ do
    $v := first(\{w \,:\, w \in V,\ \neg isMarked(w) \wedge c_H(w) = \max_{x \in V} c_H(x)\})$
    $valid := calculateValidPositions(v, \sigma^*)$
    if $\neg containingCompoundInserted(v, \sigma^*)$ then
      $\sigma^* := considerChains(v, \sigma^*)$
      $\sigma^* := insert(v, \sigma^*, valid)$
      $valid := restrictValidPositions(valid, v)$
      $\sigma^* := rollback(\sigma^*)$
      $\sigma^* := insertCompoundBorderNodes(v, \sigma^*)$
    end if
    $\sigma^* := considerChains(v, \sigma^*)$
    $\sigma^* := insert(v, \sigma^*, valid)$
    $\sigma^* := checkChains(v, \sigma^*)$
    $\sigma^* := commit(\sigma^*)$
    $mark(v)$
  end while
  $\bar{G}^* := transpose(\bar{G}^*)$
  return $\bar{G}^*$
```

As suggested above, the well-known edge crossing reduction techniques produced bad results in our experiments. Therefore, in this section, we will introduce a new technique, called the *hierarchical method*.
Instead of sorting ranks according to some heuristics, the hierarchical method starts with an empty rank structure and incrementally reinserts the nodes according to an individual complexity $c_H(v)$ for each node v. To calculate $c_H(v)$, it is sufficient to weight the number of the currently connected hierarchical edges by two and to add the number of the currently connected non-hierarchical edges. Implicitly, the hierarchical method inserts a skeleton of connected nodes and places the remaining nodes around this skeleton.
When inserting an individual node v, in the basic variant of the algorithm for each position the current

crossing number is calculated incrementally and the best position is selected. Thereby, the edge chains connected to v can be considered. If v is member of a compound, which was inserted so far, valid compound positions restrict the insertion. If the containing compound is currently not present in σ^*, first an amenable position for v is determined, then the compound valid positions are restricted to the neighbored compounds, then artifical *compound border nodes*, required for detecting compound-edge crossings, connected chains and finally v are inserted. In a postprocessing step, a transpose heuristic is applied to avoid obvious edge crossings, which may occur due to relying on local crossing numbers only. Algorithm 1 shows the main loop of the hierarchical method.
Algorithm 2 depicts the basic insertion of an individual node into its assigned rank:

Algorithm 2 insert

```
input: $v, \sigma, valid$
output: $\sigma$
  $r := r(v)$
  $\sigma_r := insertNode(\sigma_r, v, |\sigma_r| - 1)$
  $\gamma := getCrossingState(\sigma)$
  for $i := |\sigma_r| - 1$ to $0$ do
    if $valid[i] \wedge isBetter(\gamma, \sigma)$ then
      $\gamma := getCrossingState(\sigma)$
    end if
    $\sigma_r := moveNode(v, \sigma_r)$
  end for
  $\sigma := replayCrossingState(\sigma, \gamma)$
  return $\bar{G}$
```

Even if the basic variant leads to good edge crossing and compound-edge crossing numbers, a high complexity is implied: Incremental edge crossing number calculation can be implemented in $O(|V|^2)$, therefore Algorithm 2 lies in $O(|V|^3)$ and, when calculating $c_H(v)$ incrementally, Algorithm 1 in $O(|V|^4)$.
To speed up the crossing reduction method, Algorithm 2 can be replaced, e.g., by a heuristic based insertion. Algorithm 3 determines the insertion position as the barycentric position of the node to be inserted and searches for the closest valid position. Algorithm 3 can be realized in $O(|V|)$. Hence, in this variant, Algorithm 1 runs in $O(|V|^2)$.

Algorithm 3 insert barycentric

```
input: $v, \sigma, valid$
output: $\sigma$
  $pos := \frac{1}{|edges(v)|} \cdot \sum_{e=\{v,w\} \in edges(v)} \sigma_{r(w)}(w)$
  $\sigma := insertAtValidPosition(\sigma, v, pos, valid)$
  return $\bar{G}$
```

Due to the fact that the hierarchical method considers nodes with incomplete edge sets while runtime, oftentimes obvious crossing situations at neighbored nodes or chains occur. Therefore, Algorithm 1 finally calls a transpose heuristic, which is displayed in Algorithm 4.
First, similar to the transpose heuristic of the median crossing reduction in [15], neighbored nodes are exchanged if the crossing configuration can be improved. Thereby, only nodes in the same compound are considered to keep the nesting structure valid. Then, dummy nodes of chains connecting to the same compound are moved to the same side of the *compound base node*, the node, which represents the compound itself in the smallest layer number of all nodes of the compound. In particular, in UML class diagrams, as shown in Figure 1 (b), compound base nodes are oftentimes visible and require a certain area, e.g., for

representing the tab of a package. Without normalizing the dummy nodes in chains, obvious edge crossings between the chains are not detected. The final step in Algorithm 4 considers dummy nodes in chains only, recursively tries other dummy nodes in the chain to be exchanged, in particular, if the crossing configuration cannot be improved with the first dummy node exchange. Thereby, also dummy nodes next to a compound border may change their position to the opposite side of a compound to avoid edge-compound crossings.

Algorithm 4 transpose

```
input: $\bar{G} = (V, E_H, E_N, n, \sigma)$
output: $\bar{G}$
  $\bar{G} := exchangeNeighbours(\bar{G})$
  $\bar{G} := normalizeChainNodes(\bar{G})$
  $\bar{G} := exchangeChainNodes(\bar{G})$
  return $\bar{G}$
```

Obviously, even with a higher runtime complexity than the median or barycenter heuristic, the hierarchical method can also be applied to usual graphs.
In the implementation of the basic algorithm, we considered all edge crossing types except of the flat-flat case, because of an expected increase in runtime complexity. In fact, the comparison of the crossing configuration γ in Algorithm 2 is intended to handle priorities among the various crossing types, in our case, it prefers reducing hierarchy related crossings over flat related crossings. Furthermore, γ was implemented to consider the number of virtual crossings between flat edges and nodes in the same layer, i.e., the length of flat edges. Even if not shown in Algorithm 4, the exchange mechanisms also rely on the crossing configuration γ.
Unfortunately, some problems may occur, which are not sufficiently handled by all considered edge cross-

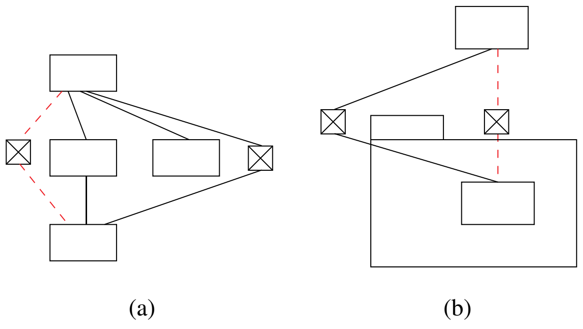

Fig. 1. Structural problems after edge crossing induced by lengthy edges which might be shortened (a) at usual nodes (b) at UML compounds. Better routes are drawn in dashed style.

ing mechanisms. In principle, the situations shown in Figure 1 can be addressed by also respecting the length of all edges in the edge crossing calculation. Currently, only pseudo-coordinates induced by the position of the nodes in their rank can be taken into account, because no coordinates are available. Furthermore, these pseudo-coordinates would need further alignment to the compound hierarchy to be considerable for general edge length calculation. Therefore, Figure 1 (a) might be handled in an improved implementation by considering general edge lengths as a secondary criterion in *isBetter*.
The situation in Figure 1 (b) can also be solved in the coordinates assignment by applying a feature, which we call *node hopping*. Thereby, selected nodes next to the compound base node, like dummy nodes, can be moved inside the compound area. In this case, the separation of tasks between edge crossing reduction

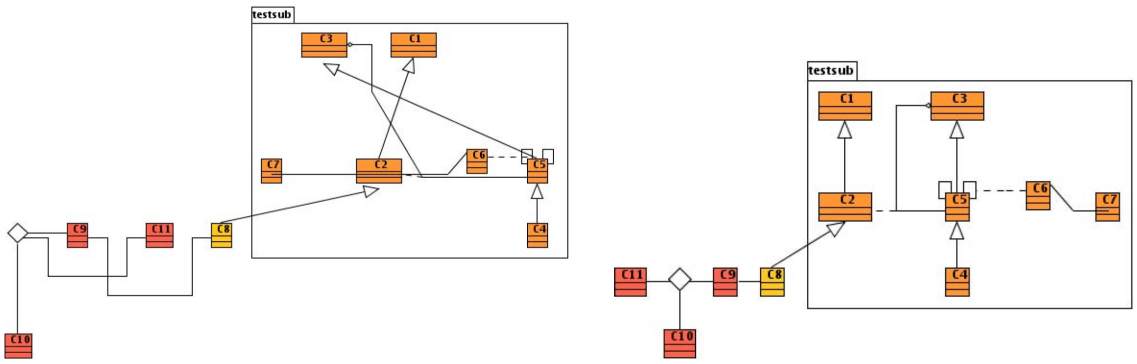

Fig. 2. The same class diagram drawn with the postprocessed barycentric algorithm (left) and the basic hierarchical algorithm (right) by *SugiBib*.

and coordinates assignment, as usual in hierarchical drawing algorithms, is not kept anymore, but a better drawing can be produced.

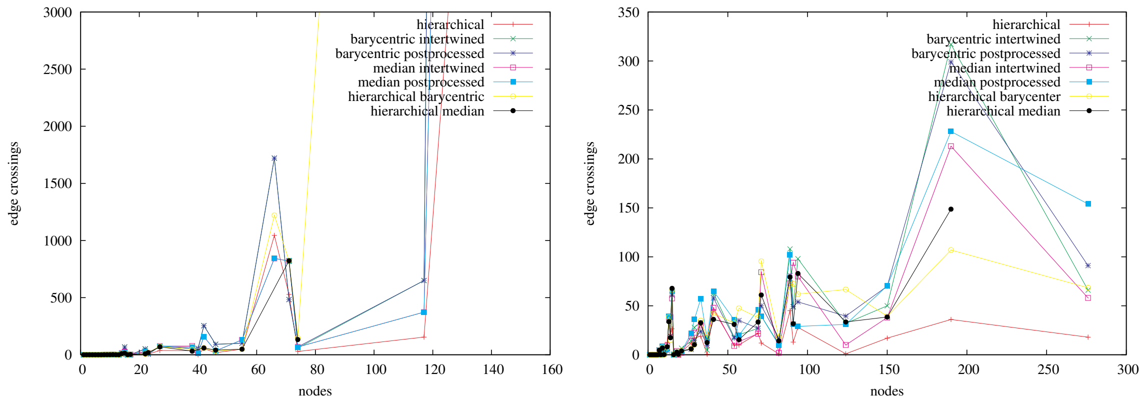

Fig. 3. Number of edge crossings of non-compound graphs (left) and compound graphs (right) in the test suite.

6 Crossing Results

We have collected crossing numbers of all hierarchical variants and the two backforced methods for barycentric and median crossing reduction mentioned in Section 2. Thereby, the number of edge crossings and the number of compound-edge crossings were measured directly after executing the individual edge crossing reduction method[1]. Furthermore, the allocated runtime was recorded for 200 diagrams (approx. 70% usual graphs) of the test suite of our layout framework *SugiBib*[2].
So far, neither the local barycenter method in [31] nor the constrained ordering in [13, 14] were considered for implementation.
The measurements have been collected running *SugiBib* on the standard JVM included in the SUN JDK 1.5.0 and a Pentium 4, 1.6 GHz with 2 GByte main memory on SuSE Linux 9.1. Example drawings taken

[1] The crossing numbers in the talk were collected after coordinates assignment and, therefore, heavily depend on the orthogonalization of non-hierarchical edges, which is currently being revised. Therefore, different values occur in this paper.
[2] `http://www.sugibib.de`

from the test suite are shown in Figure 5 and Figure 8. The measurement results are depicted in Figure 3. In most cases, the hierarchical variants produce the lowest crossing number. Even if not shown for compound graphs, the same is true for the edge-compound crossings.
To discuss the scaling of the algorithms on larger graphs, we have collected the edge crossing numbers and

crossing method	usual graph		compound graph		
	edge crossings	runtime [ms]	edge crossings	compound crossings	runtime [ms]
hierarchical (ht)	19669	2745062	59987	4849	5079148
hierarchical (m,ht)	52792	2795803	63843	6095	2463936
hierarchical (b,ht)	52792	2793299	63610	6010	2498737
hierarchical (t)	19593	2353052	59814	4842	2097709
hierarchical (m,t)	58216	500446	63479	6002	2097709
hierarchical (b,t)	58216	503134	63882	6104	2088507
barycenter (i)	90524	649570	180853	20074	7045409
barycenter (p)	90524	650439	155542	13729	7021350
median (i,t)	117368	182739	152417	20433	182739
median (p,t)	117368	183357	152417	20433	183357
median (i,ht)	106582	6037134	139534	20406	4164161
median (p,ht)	106582	6037947	139534	20406	4164779

Table 1. Runtime values of the edge crossing reduction of large UML class diagrams. (m) denotes median, (b) barycenter, (i) intertwined and (p) postprocessed variant of the individual algorithms. (t) signals that the basic transpose heuristic from [15], (ht) that the runtime-intensive transpose heuristic shown in Algorithm 4 was applied.

the edge-compound crossing numbers for an usual graph with 2504 nodes and 2959 edges and a compound graph with 100 compounds, 10710 nodes and 11794 edges. It remains unclear if UML class diagrams of this size are meaningful in practice. The results are shown in Table 1. Here, the variants of the hierarchical algorithm perform best even if a high runtime is required on the usual graph. That high runtime mainly arises from the application of the intensive transpose heuristic shown in Algorithm 4. Table 1 also displays that runtime improvements of 82.1% on usual graphs with low increases but also some improvements in crossing numbers can be reached by replacing the runtime-intensive transpose heuristic by the one described in [15]. Furthermore, in Table 1 the results of postprocessing the median ordering by Algorithm 4 are given. Here, slight improvements of the crossing numbers lead then to a high runtime.
We can conclude that on graphs of median size the basic hierarchical variant with extensive transpose heuristic appears to be the appropriate choice. On larger graphs, the basic hierarchical method with fast transpose heuristic should be considered. If then speed is more important than the number of crossings, the median variant of the hierarchical heuristic with fast transpose algorithm should be considered.
Ultimate runtime improvements without changing the implementation can be gained by generating a native executable from the Java implementation. Due to our experience, 90% runtime can be saved by applying the JET compiler[3].

7 Measuring Layout Quality

Having a set of aesthetic rules for a diagram type at hands, it is desirable to define measurement functions, e.g., to objectively measure the layout result of different tools on the same diagram. In [25], several metric formulae for general graphs have been described. Due to the fact that our aesthetic rules listed in section 4 extend existing drawing rules to also capture syntactical and semantical features of UML class diagrams, further metric formulae must be defined.

[3] `http://www.excelsior-usa.com`

Basically, we distinguish between *layout decision metrics* $m_p : G^R \rightarrow \{0,1\}$ and *layout quality metrics* $m_p : G^R \rightarrow [0\dots1] \subset \mathbb{R}$ for a diagramming rule p. Thereby, the basic rule holds that the higher the metric value, the better p is fulfulled in the drawing. A decision metric is intended to measure the conformance to severe structural rules like valid compound display, while a quality metric may capture, e.g., the percentage of avoidable bend situations. In this section, we give only a brief overview on the issues currently considered by metrics. A formal definition of the individual metrics was presented in [9].

- Conformance to the hierarchy by judging the relative distance to theoretical median positions, the number of edges running against the vertical default flow and the similarity of the shapes of the ranks in the result. Also the opposite case, the non-conformance of non-hierarchical edges, i.e., the number of edges illegally connecting nodes in ranks having no hierarhical relationships, can be taken into account.
- Closeness of the distance of association classes and comments to the connected model element(s).
- The number of edge/edge-compound crossings scaled to the number of paths in the diagram.
- The relative number of avoidable bend situations.
- The superfluous length of edges divided by the overall length of edges.
- The number of non-rectilinear edges compared to the number of non-hierarchical edges.
- Conformance to cluster containment and illegal overlappings as decision metrics.

Finally, a combined measurement is calculated by

$$m_{sqWeighted}(G^R) := \prod_{i\in M_d} m_i(G^R) \cdot \frac{\sum\limits_{i\in M_q} \gamma_i \cdot m_i(G^R)}{\sum_{i\in M_q} \gamma_i} \quad \text{with } \gamma_i := \left(\frac{\alpha_i}{\max\limits_{j\in M_q} \alpha_j}\right)^2$$

whereby M_d is the set of decision metrics, M_q denotes the set of quality metrics and α_j reflects the weight of the individual metric according to the priorities of the associated layout rule. Currently, these values are chosen according to personal preferences, experience and estimation.

Other aggregation formulae have also been tested, but none of the remiaining seemed to fit to our expectations. Furthermore, we have evaluated some layout results of the layout tools GoVisual [16] and yWorksUML [12]. Thereby, we used non-compound diagrams with simple elements to obtain a common feature set of *SugiBib*, GoVisual and yWorksUML. The results are depicted in Figure 4 and 5. Consulting the aggreated measurement values, our intuitive ranking of the results was confirmed: 0.99 was calculated for *SugiBib* and yWorksUML, 0.77 for goVisual. More details on the comparison and the results were described in [9].

Another direct application of the layout metrics is testing and debugging. By collecting the metric results of consecutive test runs over a given set of diagrams, changes in the source code of the implementation can be assessed by the metric values. This *metrics based regression testing approach* is a helpful instrument when making the decision, if new versions should be committed or revised. Unfortunately, due to the close relationship to the layout rules, sometimes conflicts between the individual metric values occur. In this case, the layout result has to be judged visually, but in most cases unique or constant values can be quantified.

8 Conclusions

We have presented a set of structural and semantical layout rules for UML and we briefly introduced the UML compliant *SugiBib* layout algorithm, which is capable of considering all these aesthetic rules. Furthermore, a new edge crossing algorithm for compound graphs was introduced, which can also be applied to usual graphs. Due to our application domain, *mixed compound graphs*, compound graphs, in which not all nodes are member of a compound, may occur. Also these graphs can be processed by the hierarchical edge crossing reduction strategy. On compound graphs, our method produces good results in reasonable runtime, on usual graphs it requires a significantly higher runtime than known-methods but also produces significantly better crossing numbers. When applying a less runtime-intensive transpose heuristic, significant runtime speed improvements with (acceptable) differences in edge crossing numbers can be measured. Therefore, depending on the desired level of quality, variants of the hierarchical methods perform best in

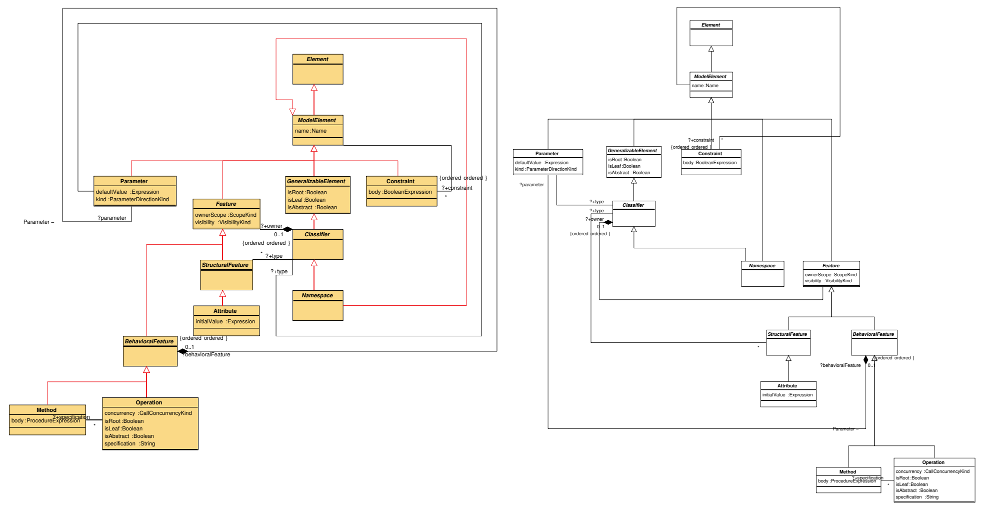

Fig. 4. A simple class diagram drawn by GoVisual (left) with problems on hierarchical edges and long edges and by yWorksUML (right).

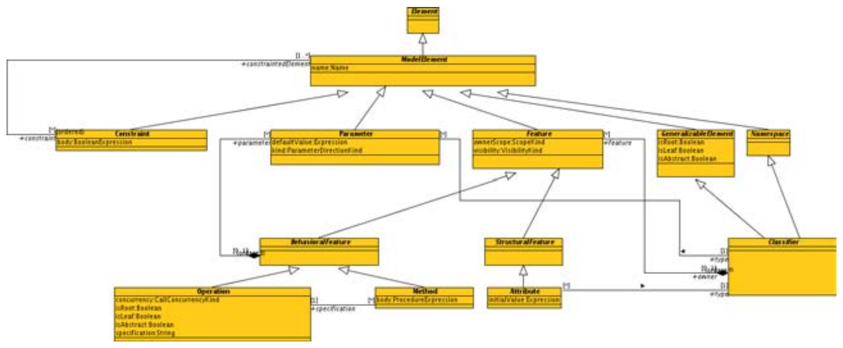

Fig. 5. Layout of the simple class diagram in Figure 4 by *SugiBib*.

most cases. Running the implementation as a native compiled executable, even for larger UML class diagrams appropriate runtimes can be achieved.
When also other quality indicators like the semantic aesthetic principles for UML class diagrams as described in [8, 9] are considered, the hierarchical variants clearly produce the best results on almost all tested graphs. This arises from the natural processing by first starting with a skeleton and then adding the remaining nodes – a technique which has been applied oftentimes by us when manually drawing UML class diagrams. Thereby, a large degree of freedom in placing nodes can be exploited.
This positive effect running the hierarchical crossing reduction can also be measured by decision and quality metrics. We have described basic measurements designed for UML class diagrams selected according to our layout rules. Beside of objective layout comparisons, layout metrics were successively applied in a metrics based regression testing approach.
In future work, also flat-flat edge crossing numbers and other heuristics to find proper insertion positions similar to Algorithm 3 may be explored. Furthermore, we are interested in validating the layout rules as well as the layout metrics by user studies and to control a evolutionary layout algorithm for UML class diagrams by the layout metrics. Both projects are currently in progress.

References

1. Andersson, E.: Automatic layout of diagrams in rational rose. Master's thesis, Computing Science Department, Uppsala University, Uppsala, Sweden (1998)
2. Bassil, S., Keller, R.K.: Software visualization tools: Survey and analysis. In: Proceedings of the Ninth International Workshop on Program Comprehension (IWPC'2001), IEEE, IEEE (2001) 7–17
3. Bernhart, M.: Semantische optimierung der anordnung von diagrammelementen in UML. Dokumentation zum Praktikum, Research Industrial Software Engineering, Institut für Softwaretechnik und Interaktive Systeme, Technische Universität Wien (2001)
4. Carpano, M.: Automatic Display of Hierarchized Graphs for Computer-Aided Decision Analysis. IEEE Transactions on Software Engineering **SE-12** (1980) 538–546
5. Eades, P., Wormald, N.C.: Edge Crossings in Drawings of Bipartite Graphs. Algorithmica **11** (1994) 379–403
6. Eichelberger, H.: Aesthetics of Class Diagrams. In: Proceedings of the First IEEE International Workshop on Visualizing Software for Understanding and Analysis, IEEE, IEEE (2002) 23–31
7. Eichelberger, H.: Evaluation-report on the layout facilities of UML tools. Technical Report 298, Institut für Informatik, University of Würzburg (2002) Institut für Informatik, University of Würzburg.
8. Eichelberger, H.: Nice Class Diagrams Admit Good Design? In: Proceedings of the 2003 ACM Symposium on Software Visualization, ACM, ACM Press (2003) 159–167
9. Eichelberger, H.: Aesthetics and Automatic Layout of UML Class Diagrams. Ph.D. thesis, Fakultät für Mathematik und Informatik, Würzburg University (2005) to appear.
10. Eichelberger, H., von Gudenberg, J.W.: On the Visualization of Java Programs. In Diehl, S., ed.: Software Visualization, International Seminar, Dagstuhl Castle, Germany, May 2001, Revised Papers. Volume 2269 of Lecture Notes in Computer Science., New York, NY, USA, Springer-Verlag Inc. (2003) 295–306
11. Eichelberger, H., von Gudenberg, J.W.: UML class diagrams - State of the art in layout techniques. In van Deursen, A., Knight, C., Maletic, J.I., Storey, M.A., eds.: Proceedings of Vissoft 2003, International Workshop on Visualizing Software for Understanding and Analysis, IEEE, IEEE (2003) 30–34
12. Eiglsperger, M., Kaufmann, M., Siebenhaller, M.: A topology-shape-metrics approach for the automatic layout of UML class diagrams. In: Proceedings of the 2003 ACM symposium on Software visualization, ACM, ACM Press (2003) 189–216
13. Forster, M.: Applying Crossing Reduction Strategies to Layered Compound Graphs. In Goodricht, M.T., Kobourov, S.G., eds.: Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers. Volume 2528 of Lecture Notes in Computer Science., New York, NY, USA, Springer-Verlag Inc. (2002) 276–284
14. Forster, M.: Crossings in Clustered Level Graphs. Ph.D. thesis, University of Passau (2004)
15. Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.P.: A Technique for Drawing Directed Graphs. IEEE Transactions on Software Engineering **19** (1993) 214–230
16. Gutwenger, C., Jünger, M., Klein, K., Kupke, J., Leipert, S., Mutzel, P.: A new approach for visualizing UML class diagrams. In: Proceedings of the 2003 ACM symposium on Software visualization, ACM, ACM Press (2003) 179–188
17. Koschke, R.: Software visualization in software maintenance, reverse engineering, and reengineering: A research survey. Journal on Software Maintenance and Evolution **15** (2003) 87–109
18. Lanza, M.: CodeCrawler - A lightweight software visualization tool. In van Deursen, A., Knight, C., Maletic, J.I., Storey, M.A., eds.: Proceedings of Vissoft 2003, International Workshop on Visualizing Software for Understanding and Analysis, IEEE, IEEE (2003) 54–55
19. Noguchi, T., Tanaka, J.: New automatic layout method based on magnetic spring model for object diagrams of OMT. In: Proceedings of International Symposium on Future Software Technology 1998 (ISFST'98). (1998) 89–94
20. Noguchi, T., Tanaka, J.: Interactive layout method for object diagrams of OMT. In: Proceedings of Asia-Pacific Software Engineering Conference (APSEC '99). (1999) 110–117
21. OMG: Model driven architecture specification (2003) Version 1.0.1.
22. OMG: Unified Modeling Language specification (2003) Version 1.5, March 2003.
23. OMG: Unified Modeling Language specification (2003) Version 2.0.
24. Purchase, H., Allder, J.A., Carrington, D.: Graph layout aesthetics in UML diagrams: User preferences. Journal of Graph Algorithms and Applications **6** (2002) 255–279
25. Purchase, H.C.: Metrics for graph drawing aesthetics. Journal of Visual Languages and Computing **13** (2002) 501–516
26. Purchase, H.C., Allder, J.A., Carrington, D.: User preference of graph layout aesthetics: A UML study. In Marks, J., ed.: Graph Drawing: 8th International Symposium GD 2000, Colonial Williamsburg, Va, USA, September 20–23, 2000: proceedings. Volume 1984 of Lecture Notes in Computer Science., New York, NY, USA, Springer-Verlag Inc. (2001) 5–18

27. Purchase, H.C., McGill, M., Colpoys, L., Carrington, D.: Graph drawing aesthetics and the comprehension of UML class diagrams: an empirical study. In Eades, P., Pattison, T., eds.: Proceedings of the Australian Symposium on Information Visualisation, Australian Computer Society, Inc. (2001) 129–137
28. Sander, G.: Layout of Compound Directed Graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken (1996)
29. Seemann, J.: Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams. In Di Battista, G., ed.: Graph Drawing: 5th International Symposium, GD'97, Rome, Italy, September 18–20, 1997: proceedings. Volume 1353 of Lecture Notes in Computer Science., New York, NY, USA, Springer-Verlag, Springer-Verlag Inc. (1997) 415–423
30. Spinellis, D.: On the declarative specification of models. IEEE Software **20** (2003) 94–96 March/April.
31. Sugiyama, K., Misue, K.: Visualization of Structural Information: Automatic Drawing of Compound Digraphs. IEEE Transactions on Systems, Man and Cybernetics **SMC-21** (1991) 876–891
32. Sugiyama, K., Tagawa, S., Toda, M.: Methods for Visual Understanding of Hierarchical System Structures. IEEE Transactions on Systems, Man and Cybernetics **SMC-11** (1981) 109–125
33. Warfield, J.: Crossing Theory and Hierarchy Mapping. IEEE Transactions on Systems, Man and Cybernetics **SMC-7** (1977) 505–523

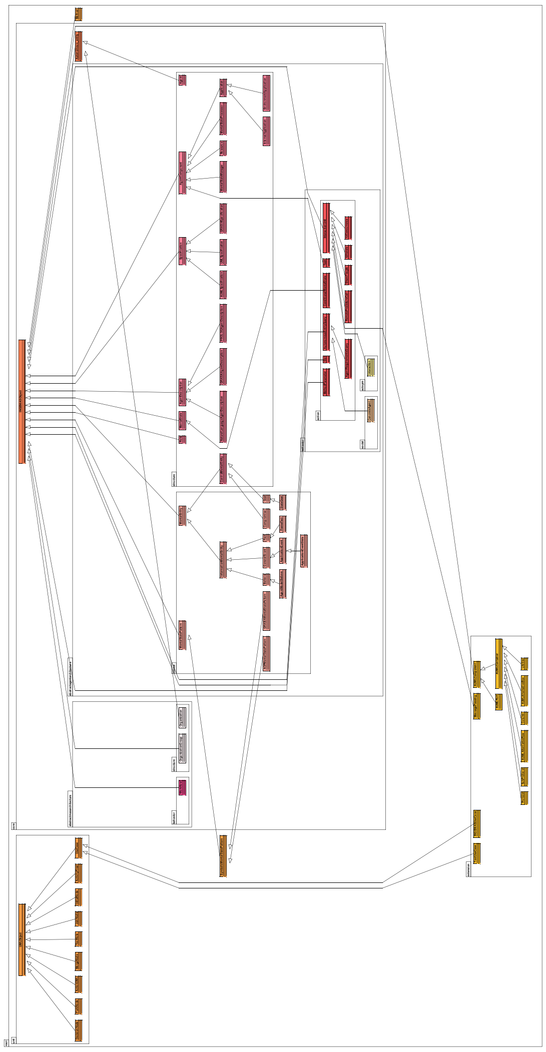

Fig. 6. An example of a compound class diagrams with 70 nodes in 12 compounds drawn by *SugiBib* applying the hierarchical edge crossing reduction.