Automatic Placement of Link Labels in Diagrams

Tobias Reinhard
Department of Informatics
University of Zurich, Switzerland
reinhard@ifi.uzh.ch

Martin Glinz
Department of Informatics
University of Zurich, Switzerland
glinz@ifi.uzh.ch

ABSTRACT

While diagrams play a central role in the software lifecycle, computer-based modeling tools are often not used in practice. At least to some extent, this is due to their lack of flexibility and support of recurring and tedious "model administration" tasks. Such tasks, like the rearrangement of existing model elements to provide the space required by a new element or the manual adjustment of lines after moving an element, distract the user from the actual modeling activities.

In this paper, we present an approach to relieve the modeler from the painful manual placement of the labels accompanying the links in a diagram by handing this task over to the modeling tool. Additionally, we give a short overview over other modeling activities that come along with the creation and manipulation of a diagram but should be handled by the tool and not the user.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Algorithms, Languages

Keywords

Modeling

1. INTRODUCTION

Diagrams show up in all stages of the software lifecycle, from the specification of requirements to the maintenance of software systems. Graphical or visual representations are often seen as more effective, especially in the communication with end users, than textual representations [10]. While there exists a multitude of tools which aim to ease the creation and manipulation of such diagrams, they are often not used in practice. In principle, a computer-based tool simplifies the work with a diagram because it can support the user in ways that are not possible on paper or white boards. However, with current modeling tools the user spends a large fraction of her/his time with "drawing" rather than modeling tasks. Overlaps between model elements have to be resolved, links have to be routed tediously around existing elements and text labels need to be placed in a way that the text remains readable. Most of the time these labor-intensive tasks distract so much from the actual modeling activity that users sidestep to more flexible approaches for the development of a model. Formal modeling tools are then used to document the final result of this process only. In order to facilitate the modeling process, tools have to relieve the users from some tasks instead of imposing additional ones on them. Ultimately, tools should make it easier and not harder to model a system.

In our previous work, we have published concepts and algorithms to manage the complexity of a diagram [14], supporting the user while changing it [15] and routing the connecting lines automatically [16]. In this paper, we present an approach to automatically place the labels that accompany the lines in a diagram. This is an additional step in evolving the interaction with modeling tools from that of simple (dumb) drawing tools towards one that really facilitates the actual modeling activities.

The remainder of the paper is organized as follows. In section 2, we briefly discuss the main interaction problems of current modeling tools and some concepts that have been proposed to alleviate them. Our label placement approach is presented in section 3. In section 4, we summarize our work and sketch some future work.

2. PROBLEMS OF INTERACTIVE EDITING

The iterative nature of the software modeling process results in specific problems because models have to be changed and updated very frequently. Each of these changes results in additional drawing tasks (e.g., rearrangement of nodes and links to provide additional or remove superfluous space). Unfortunately, most modeling tools do not support the user in handling this interaction overhead. Instead, they often employ simple, brute-force approaches (e.g., overlapping nodes, connecting nodes with straight lines or placing labels always at the middle of lines) which results in unreadable diagrams. However, a modeling tool has to offer explicit solutions for these problems, if the aim is to relieve the user from the tedious drawing tasks that are not an integral part of the actual modeling but rather supporting activities. These comprise tool-supported complexity management mechanisms, extended tool support during editing operations, a concept and algorithm that automatically routes the lines and a basic algorithm to place the labels accompanying lines automatically.

FlexiTools 2010, Cape Town, South Africa.

2.1 Complexity Management

Complexity is probably the most important challenge for the software engineering discipline. The elements of a software system interact with each other in a nonlinear fashion, so that the complexity of the whole increases much more than linearly with the number of elements [2]. The essential complexity of software systems stems largely from the complexity of the problem that has to be solved and the environment the system is embedded in [18]. As a consequence, complexity is also one of the most intractable problems of graphical representations of such systems. Unfortunately, current modeling languages do not scale [10].

Most modeling languages, including UML [11], rely on the principle of loosely coupled multi-diagram models as the primary means for separating concerns and decomposing large models. At least to some extent, this is due to the fact that current modeling tools rely on flat or practically flat models and do not provide any decent mechanisms to manage the complexity. The basic idea of our ADORA[1] approach is to reverse the underlying principles: we use an *integrated*, inherently hierarchical model instead of a loose collection of diagrams and a tool that *generates* abstractions and diagrams of manageable complexity by exploiting the hierarchy and filtering model elements. ADORA is comprised of (i) an integrated hierarchical *modeling language* [5] with hierarchical decomposition and views (structure, behavior, user interaction, etc.) and (ii) a *tool* [15] that allows a user to navigate through the hierarchy and show or hide model elements according to the selected view(s). While we show our approach in the context of the ADORA language, the presented concepts can be used for any graphical modeling language that supports a hierarchical decomposition, e.g. hierarchical UML diagrams such as component diagrams.

ADORA employs the nested box notation to represent hierarchical relationships between model elements. The representation directly suggests it's meaning [10] and facilitates a direct and straightforward fisheye zoom [4] interaction style: By *zooming-in* and *zooming-out*, the user can control the level of detail of each node. Fig. 1a) shows an abstract view of a hierarchical model: only the three top-level nodes are visible. The ellipsis after a node name is an indicator that the node has an inner structure which is hidden in the current view. By successively zooming-in nodes B and D, we get the view of Fig. 1b) which shows the details of the model in a focal point (node D) together with the global structure of the model. Conversely, by zooming-out nodes in an expanded model we get a more abstract view.

We have developed a zoom algorithm [14] which adapts the layout of the diagram in case of such zoom operations. With our fisheye zoom technique, the user can freely navigate within the hierarchical structure of a model while the algorithm solves the problem of having a user-editable layout which is nevertheless stable under multiple zooming operations. Additionally, our algorithm can be used to hide individual nodes or types of nodes to further reduce the complexity of the diagram. Thus, the user can dynamically generate different views (i.e., projections) on the model by filtering specific model elements. This view generation mechanism facilitates the integration of multiple system aspects in one coherent model while keeping the size and complexity of the resulting diagram within reasonable limits.

[1]ADORA is an acronym for **A**nalysis and **D**escription **O**f **R**equirements and **A**rchitecture

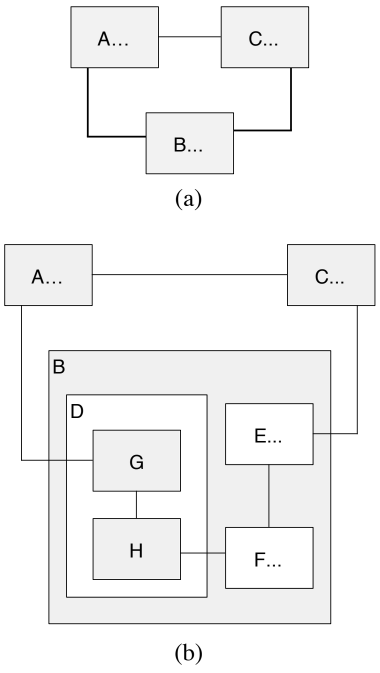

Figure 1: Fisheye Zooming

The lack of adequate complexity management mechanisms in current computer-based tools has made different authors in various fields to think about better solutions. For example, various aproaches such as SHriMP [19] or the Continuous Zoom [1] are built on the idea of fisheye views. In contrast to existing techniques, the stability of zooming and filtering operations is one of the most important properties of our approach. Furthermore, most existing techniques do not support model editing (movement, addition and removal of model elements) very well.

2.2 Editing Support

The nested box notation used to depict hierarchical relations lends itself to some kind of sophisticated tool support (like the fisheye zooming and view generation mechanism described in the previous section) but the need for tool support, especially for editing, is also significantly larger than for ordinary (flat) graphs. A change in one part of the diagram is propagated all the way up in the hierarchy until the root node is reached. The user has to manually adjust the size of all direct and indirect parent nodes and possibly the location of their siblings to provide the required space if a new node is inserted. Other editing operations such as removing or moving a node also result in a lot of tedious manual adjustment work for the user. Thus, a layout technique which automatically expands or contracts parent nodes if a node is inserted or removed is highly valuable [17] as it reduces the interaction overhead significantly. Extending the fisheye technique to support the user while editing the diagram moves it from a visualization technique to a more general "layout adjustment strategy" [19, 9]. The SHriMP approach [19] and the force-scan algorithm [9] provide some basic editing support. A major advantage of our zoom algorithm is that it can be used for automatic layout adaption if the diagram is edited for example by inserting or removing a node [14, 15].

2.3 Automatic Line Routing

The lines that are used to represent relationships between nodes in a diagram are often as important as the nodes. The effort needed to recognize relations is largely determined by the drawing of the lines in the diagram. Crossings and overlaps between lines and nodes and among different lines have a negative impact on the readability and understandability of a diagram. However, manually arranging the lines so that such overlaps do not occur burdens a lot of additional work on the user who should be released from these tedious drawing tasks. Therefore, an automatic line router that directs lines around nodes while trying to keep the lines as simple as possible is needed.

The routing of the connecting lines in a diagram has not gained much attention in the modeling field. In contrast, the routing of lines or wires is an important task in the automatic graph drawing or circuit design. However, diagrams of software systems have often different characteristics than those in these fields (e.g., nodes occupy usually no or only very little space in abstract graphs) which results in different requirements for a tool.

We have developed [16] such a routing algorithm that has the following distinctive properties: (i) it routes in real time whenever a modeler changes the layout of a model by navigating, creating a view or editing; (ii) it generates a graphically appealing layout (no collisions or overlaps, short paths) and (iii) it tries to preserve the secondary notation [13, 10] of the diagram as far as possible.

2.4 Automatic Label Placement

A problem that is closely related to the line routing one is the placement of labels that accompany lines. While line labels are not of much importance in abstract graphs, they often contain a big part of the information represented in a diagram of a software system. For example, transition descriptions in statecharts [6] or ADORA models [5] contain a large fraction of the overall information represented in the diagram. Because of this large amount of information, line labels can have a significant size.

Since the characteristics of (big) line labels are similar to those of nodes, the same problems of overlaps occur. Labels can overlap with existing nodes and/or other labels. These overlaps impede the readability of the diagram. Additionally, an inappropriate position of a line label close to other links can make it hard to determine the line it belongs to. Thus, deciding on the position of a label entails not only finding enough white space to avoid overlaps but also selecting an appropriate position relative to the line. Additionally, the position of a label relative to the line is often part of the secondary notation [13]. For example, the position of association labels in UML [11] relative to the start and end node determines the association's direction they describe.

3. LABEL PLACEMENT ALGORITHM

The problem of associating graphical features with text labels has applications in many fields such as graph drawing and cartography. Even though the "label placement problem" in cartography is NP-hard and essentially unsolved [Kakoulis and Tollis, 1997], a large number of heuristic approaches have been developed (see [3] for a survey of algorithms). While these algorithms use optimization techniques such as discrete gradient descent or simulated annealing to find a global optimum, such techniques cannot be used for an interactive modeling tool because the runtime of a label placement algorithm is a critical factor if a potentially very large number of labels have to be placed without a remarkable delay. Even though these algorithms cannot be applied directly to the modeling domain, the underlying requirements can be reused.

3.1 Label Placement Rules

Cartographers have been using rules for a good placement of labels over centuries. These rules can be used as a basis for an automatic label placement algorithm. For the problem of placing line labels in a diagram the following rules can be derived from the basic rules for placing labels on geographic maps [7, 20]:

1. Line labels must *not overlap* with other labels or other graphical features of the layout.
2. Each label can be easily *identified with exactly one line* (i.e., the assignment of a label to a line is unambiguous).
3. Each label must be placed in the *best possible position* (among all acceptable positions). The best possible position is usually defined by a set of aesthetic preferences such as that a label to the right is always preferred over one to the left of a symbol.

For diagrams that employ the nested set notation (cf. Section 2.1) an additional rule has to be added:

4. Each labels must be *fully contained* within the parent node of the line it belongs to. The parent node of a line is defined as the first common ancestor of the line's source and target node.

3.2 Tile-base Label Placement

The basic idea of the label placement approach is to use a special data structure to calculate the white space that can be used to place the labels.

3.2.1 Data Structure

The corner stitching structure [12] has originally been developed as an efficient storage mechanism for VLSI layout systems and has two important features:

- All space, whether occupied by a node or empty, is explicitly represented in the structure. This explicit representation of the empty space makes it possible to provide fast geometrical algorithms to locate the space that is available for the placement of labels.
- The space is divided into rectangular areas that are stitched together at their corners like a patchwork quilt. These corner stitches allow easy modifications of the structure and lead to efficient implementations of a variety of geometric operations.

The corner stitching structure for the four nodes of Fig. 2 is represented by the dashed lines. The space is divided into a mosaic with rectangular tiles of two types: *space tiles* and

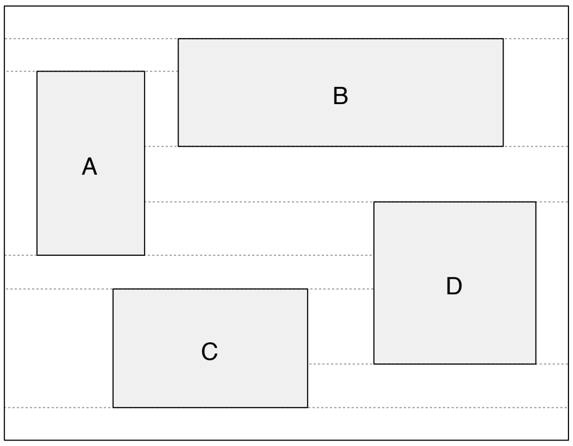

Figure 2: Corner stitching structure

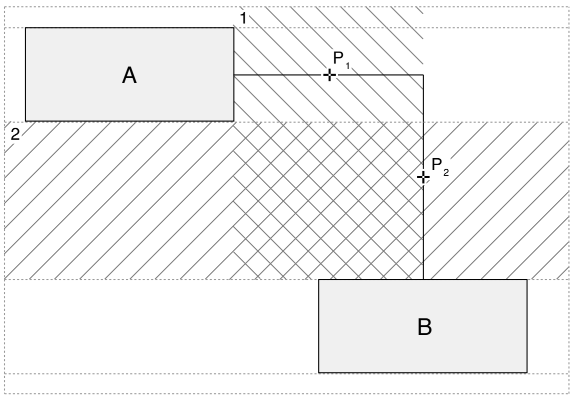

Figure 3: Calculation of the label space on the corner stitching structure

node tiles. Tiles must be rectangles with sides parallel to the X and Y axes. The only constraint is that nodes must not overlap. The space tiles are organized as maximal horizontal strips: no space tile ever has another space tile immediately to its right or left. However, there can be another space tile directly above or below a space tile. This organization ensures that there is exactly one decomposition of the empty space into space tiles for each arrangement of node tiles. Additionally, it results in a clear upper bound for the number of tiles: In a diagram with n nodes, there will never be more than $3n + 1$ space tiles (see [12] for a proof). Initially, we have used the corner stitching data structure as the basic data structure for our line routing algorithm [16].

3.2.2 Algorithm

The calculation of the white space becomes trivial because the empty space is explicitly represented in the corner stitching structure. The white space calculation avoids overlaps of line labels with nodes, existing lines (if they are explicitly represented in the corner stitching structure [15]) and existing line labels. Additionally, the use of the corner stitching structure results in labels that are always contained within the line's parent node. A set of potential label positions with a predefined relative position to the line is then calculated within this white space. Among these candidate positions the best one with respect to some predefined criteria is selected. The detailed steps of the algorithm are the following:

1. Iterate over all (vertical and horizontal) line segments.

2. Use the corner stitching structure to calculate the available free space around the segment: First a reference point for each label is defined. This point can either be the midpoint of the segment if the label should be shown close to the middle of the line or the segment's start or end point. The corner stitching structure is then used to find the tile this reference point lies in (see [12] for the detailed algorithm to find a tile at a given location). Finally, the free space is calculated by vertically extending the boundaries of the tile containing the reference point. Fig. 3 shows an example of this step. P_1 and P_2 are the reference points for the two line segments of the line connecting nodes A and B.

   The corner stitching structure for this constellation is shown by the dashed lines. The hatched areas 1 and 2 show the free space that has been calculated for P_1 and P_2 respectively. In our current implementation the width of the free space for horizontal line segments is restricted to the length of the segment (as shown for the hatched area 1 in Fig. 3). The vertical expansion of the free space stops as soon as the width of the next tile above or below is smaller than the current width of the white space. That's why the expansion of the free space 2 stops below node A and above node B.

3. Determine three candidate positions for each line segment: For a horizontal segment these are above, below and vertically centered around the reference point. For a vertical segment they are to the left, the right and vertically centered around the reference point.

   Fig. 4 shows the six candidate positions for a line label that belongs to the line between nodes A and B. In our current implementation the label is asked for its required height given a specific maximal width. This maximal width is directly taken from the width of the free space adjusted by the relative position of the label (i.e., the maximal width of label 6 in Fig. 4 is the width of the free space to the right of the second line segment). This is a very basic approach which tries to make labels as wide as possible. Other more sophisticated approaches that take for example the content of the label into account to calculate its boundaries can easily be implemented as part of the label's internal size calculation without changing the placement algorithm. For example, in statecharts such an approach can always try to write each of the components of a transition description on a separate line.

4. Select one of these candidates: To do so we associate a score with each candidate position. This score combines a set of a priori preferences with a value that reflects the severity of the violation of the basic label placement rules (1) and (2) of section 3.1. The basic label placement rules can be violated if there is not enough space for the label at the given position so that it overlaps with nodes (or other labels) or if the label is crossed by the line it belongs to.

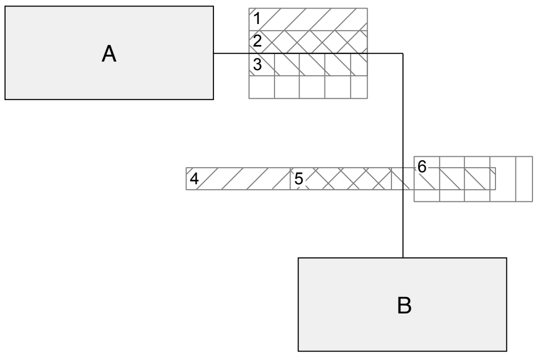

Figure 4: Candidate label positions

For the example of Fig. 4, the algorithm selects label position 1 if it prefers labels above horizontal segments or position 4 if it prefers labels to the left of vertical segments. The scoring can, together with a specific reference point, also be used to prefer positions close to the source or target node if such a position is an important part of the secondary notation.

4. CONCLUSIONS

We have presented an approach to automatically place link labels in a diagram. The presented algorithm has been implemented as part of the ADORA tool. Together with our previous work on complexity management in diagrams, tool-supported editing and line routing, this technique brings us one step further towards our goal of relieving the user from the tedious drawing tasks that are not an integral part of actual modeling activity.

Much more sophisticated techniques can be built on top of our basic label placement approach. The three main extension points are the calculation of the white space (e.g., calculating the free space for each candidate position independently), the calculation of the candidate positions (e.g., using multiple reference points or different label boundaries for each segment) and the scoring of the candidate positions. However, already the presented simple approach can mitigate the label problem to a large extend even though it may not always find the "best" position for a label.

5. REFERENCES

[1] L. Bartram, A. Ho, J. Dill, and F. Henigman. The Continuous Zoom: A Constrained Fisheye Technique for Viewing and Navigating Large Information Spaces. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST '95)*, pages 207–215, 1995.

[2] F. P. Brooks. No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[3] J. Christensen, J. Marks, and S. Shieber. An Empirical Study of Algorithms for Point-Feature Label Placement. *ACM Transactions on Graphics*, 14(3):203–232, July 1995.

[4] G. W. Furnas. Generalized Fisheye Views. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 16–23, 1986.

[5] M. Glinz, S. Berner, and S. Joos. Object-oriented modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

[6] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[7] E. Imhof. Positioning Names on Maps. *The American Cartographer*, 2(2):128–144, 1975.

[8] K. G. Kakoulis and I. G. Tollis. An Algorithm for Labeling Edges of Hierarchical Drawings. In *Proceedings of the 5th International Symposium on Graph Drawing*, pages 169–180, 1997.

[9] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[10] D. L. Moody. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, November/December 2009.

[11] OMG. Unified Modeling Language: Superstructure Version 2.0. Document formal/05-07-04, Object Management Group, 2005.

[12] J. K. Ousterhout. Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 3(1):87–100, 1984.

[13] M. Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38(6):33–44, June 1995.

[14] T. Reinhard, S. Meier, and M. Glinz. An Improved Fisheye Zoom Algorithm for Visualizing and Editing Hierarchical Models. In *Proceedings of the Second International Workshop on Requirements Engineering Visualization (REV'07)*, 2007.

[15] T. Reinhard, S. Meier, R. Stoiber, C. Cramer, and M. Glinz. Tool Support for the Navigation in Graphical Models. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 823–826, 2008.

[16] T. Reinhard, C. Seybold, S. Meier, M. Glinz, and N. Merlo-Schett. Human-Friendly Line Routing for Hierarchical Diagrams. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 273–276, 2006.

[17] C. Seybold, M. Glinz, S. Meier, and N. Merlo-Schett. An Effective Layout Adaptation Technique for a Graphical Modeling Tool. In *Proceedings of the 25th International Conference on Software Engineering*, pages 826 – 827, 2003.

[18] H. A. Simon. *The Sciences of the Artificial.* The MIT Press, Cambridge, MA, third edition, 1996.

[19] M.-A. D. Storey and H. A. Müller. Graph Layout Adjustment Strategies. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 487–499, 1995.

[20] P. Yoeli. The Logic of Automated Map Lettering. *The Cartographic Journal*, 9(2):99–108, 1972.