A Novel Algorithm for Line Routing in Hierarchical Diagrams

Tobias Reinhard, Christian Seybold, Silvio Meier, Martin Glinz, Nancy Merlo-Schett
Requirements Engineering Group, Department of Informatics, University of Zurich
Binzmuehlestr. 14, CH-8050 Zurich, Switzerland
reinhard, seybold, smeier, glinz, schett @ifi.unizh.ch

Technical Report ifi-2006.08

Abstract

Hierarchical diagrams are well-suited for visualizing the structure and decomposition of complex systems. With the advent of UML 2.0, in particular the new composite structure diagram, hierarchical models have entered the modeling mainstream.

However, the current tools poorly support hierarchical modeling and visualization. Simple explosive zooming is the most common means for navigating through hierarchies; some tools even visualize the complete hierarchy in a single large diagram. The line routing algorithms used by the current tools are poorly suited to this task: for example, they produce lines that run across nodes or overlap with other lines.

In this paper, we present a novel algorithm for line routing in hierarchical models which, together with our previous work on node positioning, yields visualizations of hierarchical models that can easily be browsed and edited. In particular, our algorithm (i) produces an esthetically appealing layout, (ii) routes in real-time, and (iii) preserves the secondary notation of the diagrams as far as possible.

1 Introduction

With the advent of UML 2.0 [18], there has been renewed interest in hierarchical models and their effective visualization. The traditional way of visualizing hierarchically structured models is by explosive zooming: when looking at the details of a particular node, the diagram with the details either replaces the previously displayed diagram or it is opened in a new window that supersedes the previously viewed one. In both cases, the context of the zoomed node is lost: with explosive zooming, one cannot view the details of a node and its context in the same diagram. For human understanding of diagrams, this is bad because humans typically want to see their focus of interest in detail together with its surrounding context.

In our research group at the University of Zurich, we have investigated hierarchically structured models and their visualization in the framework of the ADORA project since 1998 and have developed both a hierarchically structured modeling language and a prototype tool for visualizing such hierarchies [12, 24].

With hierarchical models, we typically want that tools display not only the views that have been drawn by the modeler, but exploit the full power of such models by allowing arbitrarily navigation and zooming in the models. In particular, abstraction and filtering capabilities are required. Abstractions help modelers concentrate on their focus of interest by hiding all lower levels of the hierarchy. Filtering mechanisms display only those model element types that the modeler is currently interested in and hide all others.

Designing tools with such capabilities is a non-trivial task because

- in order to support abstraction and filtering, the tool must be able to generate dynamically different views from an abstract internal representation of the model,
- generated layouts should resemble the original layout drawn by the modeler as far as possible.

The second point is important because a modeler builds a cognitive map [1] of locations and shapes of objects and lines when creating models and later browsing or modifying them. This layout information that helps modelers read and comprehend diagrams, is called secondary notation [21]. In order to make generated views easily comprehensible for humans, a tool must preserve the secondary notation as far as possible when generating a view or when navigating from a given view to a more abstract or a more detailed one.

However, with the requirement of preserving the secondary notation, the task of view generation becomes difficult: the well-known algorithms for graph layout or VLSI layout, for example [6], cannot be used because they generate the layout from scratch and ignore the secondary notation. We have surveyed several UML tools (see Appendix)

and found that none of them solves the problem of preserving the secondary notation in a satisfactory way.

The layouting problem can be roughly divided into (a) positioning of nodes (classes, states, activities, etc.) and (b) routing lines that connect nodes (associations, state transitions, port connections, etc.).

In our previous work on the ADORA tool [24], we have developed node positioning algorithms for editing, navigating and zooming hierarchical models such that the secondary notation is preserved. In order to make this paper more self-contained, we briefly survey these algorithms in Section 3.

The main contribution of this paper is a novel algorithm for routing lines between nodes in hierarchically structured models that

- routes in real-time when a modeler navigates, zooms or edits a view,
- generates a graphically appealing layout (no collisions / overlaps between lines, short paths),
- preserves the secondary notation (i.e. manual adjustments of the lines) as far as possible,
- allows a modeler to modify the generated layout of a line route and preserves this modification as a new secondary notation in subsequent view modifications.

All presented algorithms for node positioning and line routing have been implemented in our ADORA tool [24]. However, our algorithms work on any hierarchical box-and-line language, for example, UML 2.0 composite structure diagrams, activity diagrams or state machine diagrams.

The remainder of the paper is organized as follows. In Section 2, we briefly survey the capabilities of current hierarchical modeling tools with respect to layout generation and preservation of secondary notation. Section 3 explains our algorithms for node positioning in hierarchical diagrams. Section 4 is the core of this paper, where we describe our line routing algorithm. In Section 5, we apply our zoom and line routing algorithm implemented as an eclipse plugin to a sample model of a heating control system. Related work is discussed in Section 6. In Section 7, we summarize our results, discuss advantages and limitations and sketch our future work.

2 Capabilities of Current Hierarchical Modeling Tools

We surveyed about twelve tools with respect to their ability for visualizing, navigating and editing hierarchical models. Most of them are UML tools. We concentrated on those diagram types that allow hierarchical decomposition, in particular composite structure diagrams, state diagrams and activity diagrams.

In our opinion, the minimum features that a hierarchical modeling tool should support are: smart zooming and navigation operations for browsing through the hierarchy, occlusion free positioning of nodes in decomposition hierarchies, automated line routing, and adequate mechanisms for filtering and abstraction. All operations on the models should preserve the secondary notation as far as possible. Additionally, smart editing functionality would be valuable for avoiding tedious manual re-layouting work when inserting or deleting model elements. None of the investigated tools has all these features. The details of our tool survey can be found in the appendix.

3 Node Positioning in Hierarchical Model Views

Dynamic layout algorithms are required to adequately handle large, hierarchically nested diagrams. Conventional tools without dynamic layout support either statically display the whole diagram including all details or show sub hierarchies in a linked diagram replacing or hiding the current content. In both cases, the global context of the visible model part is lost.

In contrast to this, we visualize the whole model in a single window and apply layout algorithms to adapt the desired level of detail. This allows to display local detail where needed while keeping the global context visible. We use a logical fisheye algorithm [11] to perform the rearrangement of the diagram elements dynamically while the secondary notation is preserved as far as possible.

Variants of our logical fisheye algorithm [24, 12, 3] perform different tasks. We use it (i) for zooming-in and -out single objects to reveal or hide details, (ii) together with insertion or deletion operations it generates the necessary space or removes the free gaps, and (iii) applied on all objects of a certain type, it can be used as a filter to hide or re-display objects. All three variants rely on the same algorithmic idea.

The idea is to move surrounding objects radially away on a space request or contract them radially on freed space. This is schematically shown in Fig. 1. Object A is being zoomed-in and therefore has to be expanded from its old size (solid) to its new size (dashed). The surrounding objects B, C, and D would get too close or even overlap with A's new size. To shift them away, the vectors $\vec{V_B}$, $\vec{V_C}$, and $\vec{V_D}$ are calculated by connecting each object's center to the center of A. The length of the vector is determined by the expansion size of A in that direction. For a contraction, the calculations are reversed correspondingly. The radial shifting assures stable, relative positions among the objects B, C, and D and to object A.

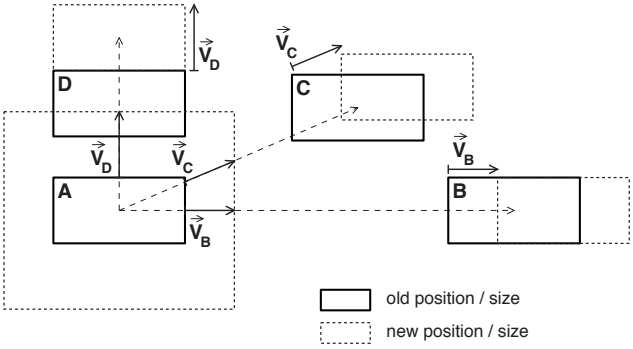

Figure 1. Zooming-In: surrounding objects are shifted away

In a hierarchical diagram, the objects are embedded in a parent container P. On an expansion or contraction, P's size has to be adapted accordingly. The secondary notation in form of P's shape is preserved by keeping the distance between the children and its surrounding container invariant.

Adapting P's size requires layout adaptations on the next higher hierarchy level. P's siblings in turn will be radially contracted or shifted away. P's parent container again has to be adapted in size. We apply our algorithm recursively up to the root object.

Whenever the layout of a diagram is changed by zooming or editing nodes, the lines connecting the nodes in the diagram must be re-routed. In the following section, we present our novel line routing algorithm that is smart with respect to avoiding occlusions and overlaps and is able to preserve user-defined secondary notation.

4 Line Routing for Hierarchical Elements

The lines that represent relationships in a diagram are usually added in a second step after the placement of the nodes. To maintain the readability of the diagram it is desirable that a line between two nodes does not pass through any other nodes or overlap with other lines. The cumbersome line routing task should be done automatically to let the user focus on his primary goal of creating a diagram.

The application of a zoom algorithm on the hierarchical structure leads to a dynamic layout of the diagram, i.e. the layout is not only changed by editing operations but also by the logical navigation. Changes of the layout demand an adjustment of the lines that has to be done automatically and in real-time. Otherwise the user would spend most of his time in rerouting the existing lines. This dynamic layout leads to the additional requirement that the runtime complexity of the line routing algorithm has to permit an interactive usage.

In contrast to techniques from other domains such as VLSI design, esthetics play the more important role in diagrams than e.g. physical constraints. The lines in a diagram have to be easy to follow and add clear meaning to the diagram. But there is only little information available about what is a "good" line [22]. In most areas that use diagrams to visualize information, a preferred style or common sense has emerged, e.g. rectilinear lines in circuit diagrams.

Lines in a hierarchical diagram often connect nodes that are located in different branches of the hierarchy tree that is visualized by the diagram. It is therefore not always obvious which nodes are potential obstacles for a line. The gray shaded nodes in Fig. 2 are the potential obstacles for the line that connects the nodes 1.1 and 4.2. These nodes lie on different levels of the hierarchy (e.g. nodes 1.2 and 3 are both obstacles even though node 1.2 is one hierarchical level below node 3). The potential obstacles of a line have therefore to be computed by traversing the hierarchy tree from the root node the line is completely contained in (node 0 in Fig. 2), to both the source and target node of the line. Thereby, all the nodes on each level are collected that are not contained in the path from the source or target node to the root node. When the obstacles are computed, the hierarchical line routing problem becomes an instance of the usual "flat" line routing problem.

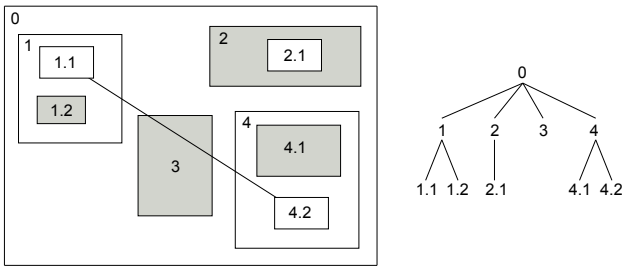

Figure 2. Line routing in a hierarchical structure

We are focusing on diagrams that are incrementally created by a human user and not on diagrams that are automatically created to visualize an existing structure. The lines therefore have to be routed sequentially, each line when it is added to the diagram. As the position and the size of the nodes define the secondary notation, the line routing algorithm must not change the layout of the diagram.

4.1 The routing problem

If we route the lines, as an initial simplification, independently (i.e. without any regard to existing lines) around the nodes that represent obstacles, we are faced with an instance of the routing problem that arises in different domains.

The routing problem has been studied extensively in VLSI design to automatically layout the wires that connect the circuit components. The first and perhaps best known routing algorithm for VLSI design is Lee's algorithm [15],

which is an application of Dijkstra's breadth-first shortest path search algorithm [4] to a uniform grid. Lee's algorithm is based on the expansion of a diamond-shaped wave from the source point that continues until the target point is reached. The algorithm always finds a solution if one exists, and ensures an optimal solution. The major drawback of this approach is that its space and runtime complexity is $O(mn)$ for a grid with $m * n$ cells.

Fig. 3 shows the proceeding of Lee's algorithm after the sixth iteration of the expansion phase while routing a line from the source cell s to the target cell t. The arrows indicate for which cells the distance values are calculated in the next step. The shortest path can be found after the expansion phase by starting at the tile t and moving always to one of the neighboring tiles that has a lower cost value than the current tile.

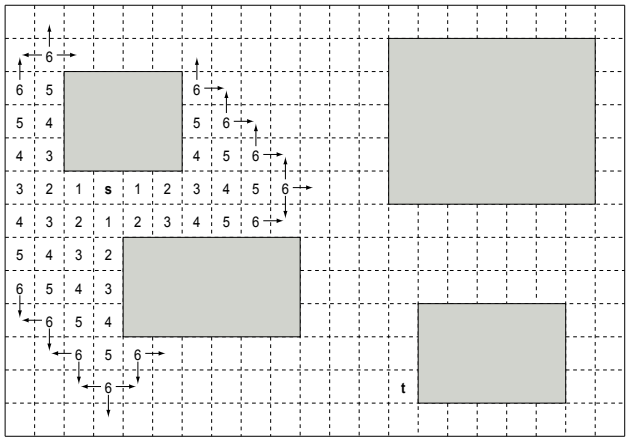

Figure 3. Lee's algorithm

4.1.1 Data structure

Instead of a uniform grid we use the corner stitching structure [20] as the underlying data structure for the line routing. The corner stitching structure has originally been developed as an efficient storage mechanism for VLSI layout systems and has two important features:

- All the space, whether occupied by a node or empty, is explicitly represented in the structure.
- The space is divided into rectangular areas that are stitched together at their corners like a patchwork quilt.

Fig. 4 shows four nodes represented in the corner stitching structure. The space is divided into a mosaic with rectangular tiles of two types: space tiles and solid tiles. The space tiles are organized as maximal horizontal strips, i.e. no space tile ever has another space tile immediately to its right or left.

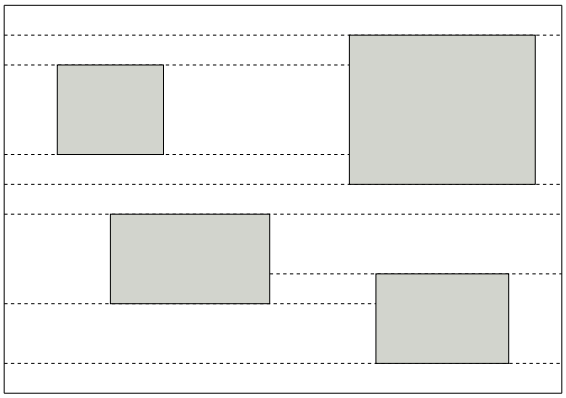

Figure 4. Corner Stitching data structure

The organization as maximal horizontal strips results in clean upper bounds on the number of space tiles independently of the layout of the nodes and the overall size of the diagram. In a diagram with N nodes, there will never be more than $3N + 1$ space tiles (for a proof of this limit see [20]).

The corner stitching structure in its initial form is restricted to rectilinear geometries, because the tiles have to be rectangles. This is no real constraint for our diagrams, because we use rectangles to visualize the nodes. If the shapes that have to be visualized are not rectangles, the corner stitching structure can be extended, e.g. to use trapezoids instead of rectangles [16].

The corner stitching structure provides a variety of operations, such as neighbor-finding, point-finding, stretching and compaction. The algorithms for these operations depend only on local information (i.e. the objects in the immediate vicinity of the focus of the operation). The expected runtime is therefore generally linear to the number of nearby objects. The most important features of the corner stitching structure in connection with line routing are the explicit representation of white space that can be used for line routing and the easy determination of all the neighbors of a tile.

4.1.2 White space computation

Our line routing approach is divided into two completely decoupled steps: The first step computes one or multiple sequences of space tiles through which the shortest path has to pass. The second step computes the line itself, i.e. the bend points in a polyline or the curves of a spline inside the white space tiles that have been calculated in the first step. These two steps are described in the current and the next sub-section.

We apply the fundamental wave expansion idea of Lee's algorithm to the corner stitching structure instead of a uniform grid structure to compute the path(s) of space tiles through which the line has to pass. The corner stitching

structure has a clear upper limit on the number of tiles depending only on the number of nodes, whereas the number of cells in a uniform grid is determined by the size and resolution of the grid. Diagrams (especially hierarchical structured diagrams) occupy usually a lot of space while the number of nodes remains small. Integrated circuits have the opposite characteristics: the circuit board has to be as small as possible (highly integrated) and often contains up to millions of components. So we are applying Lee's algorithm to a data structure that is better suited for the domain of diagrams than the algorithm's initial domain of integrated circuits.

During its expansion phase, our algorithm computes a distance value for the space tiles of the structure. Due to the non-uniform tile size of the corner stitching structure, it is not possible to use the distance from the source point to the tiles as distance values, because there may be multiple different values for one tile. We therefore use a combination of the source distance and the target distance which are both measured in the Manhattan distance[1]. Furthermore, the algorithm computes for each tile the point P inside the tile, where the distance is actually measured. For the actual search, an ordered data structure (e.g. heap, priority queue) denoted as Ω is used. Below, we present an informal description of the algorithm:

1. Construct the corner stitching structure and determine the tile T_{start} that contains the source point s and the tile T_{end} that contains the target point t.

2. Set the point P for T_{start} to the source point s, the source distance of T_{start} to 0 and the distance of T_{start} to the Manhattan distance between P and the target point t. Insert T_{start} into Ω.

3. As long as Ω contains tiles: Remove the tile T with the lowest distance value from Ω. If this tile is the tile T_{end}, a path has been found. Otherwise, calculate for each neighboring space tile T_{next} the point P_{next}, i.e the point inside T_{next} closest to the point P of the current tile T. Compute for each of these neighboring tiles two distance values: The *source distance* σ is calculated by adding the Manhattan distance between the point P and the point P_{next} to the source distance of T. The *distance* δ is calculated by adding the Manhattan distance between P_{next} and the target point t to σ. Equations 1 and 2 show the calculation of the source distance σ and the distance δ for the tile T_{next} relative to the tile T, whereas $\lambda(P_1, P_2)$ denotes the Manhattan distance between the points P_1 and P_2:

$$\sigma(T_{next}) = \sigma(T) + \lambda(P, P_{next}) \quad (1)$$

$$\delta(T_{next}) = \sigma(T_{next}) + \lambda(P_{next}, t) \quad (2)$$

   Check whether the calculated distance value for T_{next} is lower than a previously calculated value for this tile. If so, update the distance δ and source distance σ values of T_{next}. Insert the tile T_{next} into Ω.

4. The sequence(s) of space tiles that constitute the shortest path can be found, in analogy to Lee's and Dijkstra's algorithm, by moving from a tile to the neighbors that have the same distance value δ. If multiple neighbors have the same value, there exist multiple solutions and the current tile is a branch.

Fig. 5 shows the corner stitching structure after the second iteration through step 3 of the algorithm. The current tile T corresponds to T_6 and the distance values σ and δ for the tiles T_4, T_5 and T_7 are calculated. According to equation 1, the source distance σ for T_7 corresponds to the addition of the source distance of T_6 (which is zero in this case) to the Manhattan distance between the points P and P_{next} in T or T_{next}, respectively, which is 36. The distance δ is calculated by adding the Manhattan distance between the point P_{next} and the target point t, which is 576, to the source distance σ.

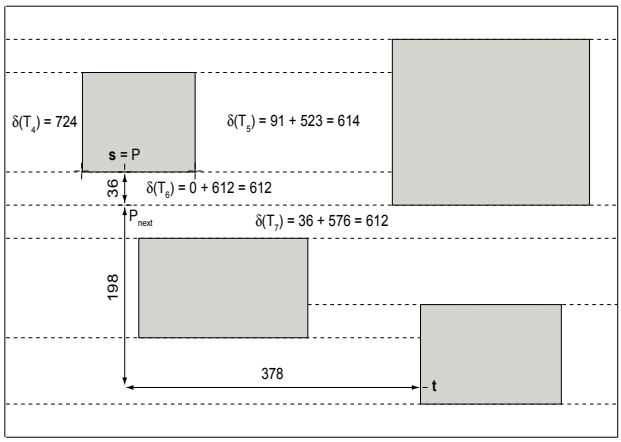

Figure 5. Routing algorithm

4.1.3 Line routing

As the algorithm for finding the space tiles that the shortest path has to pass through is decoupled from the algorithm that does the actual routing, i.e. determine the informations that are necessary to draw the line, different algorithms that produce different styles of lines can be implemented on top of the algorithm described in Section 4.1.2. Fig. 6 shows three different line routing styles: a rectilinear polyline, an unconstrained polyline and a spline.

We have currently implemented a rectilinear line routing style because this style nicely fits the general diagram layout that uses rectilinear nodes. Furthermore, it is straightforward and easy to implement due to the rectilinear nature of the tiles in the corner stitching structure.

[1]The Manhattan distance, also known as the L_1-distance, between two points P and Q is defined as the sum of the lengths of the projections of the line segments onto the coordinate axes: $\lambda(P, Q) = |x_P - x_Q| + |y_P - y_Q|$

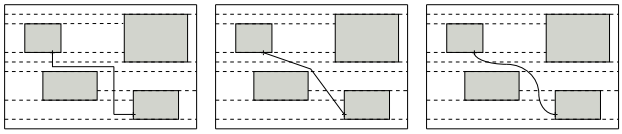

Figure 6. Different line routing styles

4.2 Line crossings

As long as the lines are routed independently from each other, it is impossible to reduce line crossings or avoid the overlapping of line segments. But line crossings and overlappings reduce the readability of a diagram dramatically, because it becomes hard to follow the lines and to find out which nodes are connected. To maintain a good readability of the diagram, it is therefore unavoidable to take existing lines into account while routing a new line.

It is possible to route the lines independently and reduce line crossings afterwards by explicitly checking for intersecting lines and move the lines or line segments to reduce crossings or avoid overlappings. However, this solution can only provide local optimizations because the overall path is already defined. It is therefore not possible to avoid areas that are already occupied by a lot of lines by looking for a detour through areas that are occupied by less lines.

By extending the corner stitching structure and the algorithm of Section 4.1.2, we can consider global properties, like avoiding line crossings, during the routing of a line. We extend the corner stitching structure by a third tile type: the line tile. Line tiles are space tiles weighted by a constant *cost factor* α. The algorithm now calculates cost values instead of distance values for the tiles. A higher cost factor α of a tile increases the costs for a line to pass through this tile. We are therefore transforming the source distance σ and the distance δ into a *source cost* ω and a cost γ. Equations 3 and 4 show the extension of equations 1 and 2 with the cost factor α:

$$\omega(T_{next}) = \omega(T) + \alpha * \lambda(P, P_{next}) \quad (3)$$

$$\gamma(T_{next}) = \omega(T_{next}) + \lambda(P_{next}, t) \quad (4)$$

Fig. 7 shows a snapshot after the third iteration through step 3 of the algorithm of Section 4.1.2 extended by the cost factor, while routing a line from the source point s to the target point t. The cost γ for the tile T_8 has been calculated during the first iteration. The second iteration lead to the costs for tiles T_4, T_9, T_{10} and T_{13}. The cost for tile T_{11} is increased by a cost factor α of 4 because tile T_{10} is a line tile. Therefore, the Manhattan distance λ between the point P in T_{10} and the point P_{next} in tile T_{11} is multiplied by the cost factor of the line tile T_{10}.

The corner stitching structure has to be extended by an additional invariant in order to make the described algorithm work in all situations as expected: A line tile can never have multiple space tiles as left or right neighbors, i.e. if a line tile touches two or more space tiles on the left or right side, it has to be split accordingly. That is the reason why there are four line tiles in Fig. 7 instead of only one that covers the whole existing line. As noted in Section 4.1.1, the corner stitching structure in its initial form is restricted to a rectilinear geometry, and the technique to avoid line intersections can therefore only be applied to rectilinear lines[2].

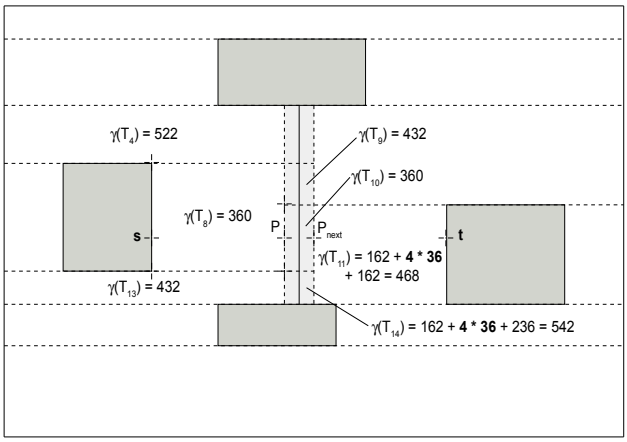

Figure 7. Weighted tiles

Step 4 of the algorithm in Section 4.1.2 has to be extended so that the neighbors with the same *or a lower* cost value are selected during the retracing. The cost value becomes lower, if a line tile is crossed during the retrace.

The cost factor α offers a possibility to the user to influence the line routing of a diagram. By setting α for the line tiles, the user can express the length of the detour s/he is willing to accept to avoid a line crossing. The idea of weighted tiles is not restricted to line tiles. It may be extended to avoid intersections with labels or to indicate preferred areas for the routing.

4.3 Preserving Secondary Notation

In connection with line routing, the secondary notation has two aspects: The routing algorithm has to tolerate some user influence, i.e. the line should not be routed completely automatically. And once defined, the secondary notation of the lines has to be preserved in the case of layout changes e.g. produced by the zooming algorithm.

The algorithm that has been described so far, lets the user select where the source and target points are anchored on the source and target node. Furthermore, by varying the cost factor α, the user can define to what extent the algorithm should avoid line intersections. If there exist multiple

[2]The approach is actually not restricted to the corner stitching structure. It can be used for any geometric structure that provides informations about the white space in the diagram.

shortest paths, it is also possible to let the user select one solution manually or let him/her provide a selection function to the algorithm.

But the definition of the secondary notation by the user does not have to end with the application of the routing algorithm. The user can change the line that has been proposed by the routing algorithm by moving the segments of a rectilinear line or the bending points of a unconstrained polyline. These changes have to be preserved when the line has to be rerouted in case of a layout modification. The corner stitching structure is a valuable source of information for preserving the secondary notation, because it permits an easy extraction of geometric informations (e.g closest neighbor or available free space) about the layout of the diagram. It is therefore possible to let the user move the line segments in Fig. 8 closer to the node that the line has to be routed around, store this information (symbolized by the arrows in the figure) and restore the line after a layout modification has occurred. Fig. 9 shows a screenshot of the ADORA modeling tool during a zoom operation while preserving the secondary notation of the lines.

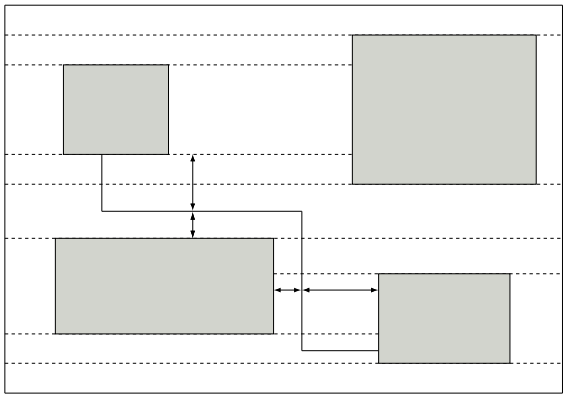

Figure 8. Preserving the secondary notation

It is usually necessary to use the routing algorithm to reroute the lines completely after a layout modification, because additional segments or bending points may be necessary or existing segments may be no longer needed. The routing algorithm can therefore not always guarantee that the lines look still the same after a layout modification, because the former routing may have become absurd in the new layout.

4.4 Performance Analysis

We have done a short performance analysis to test whether our routing algorithm can be used in an interactive environment. The results show that the algorithm is able to route a large number of lines fast enough to avoid a user irritation. The setup and detailed results of the performance analysis can be found in the appendix.

5 Example Application

We have implemented our approach as an Eclipse plugin [19] that allows to create models based on our ADORA language. Furthermore, the presented zoom and line algorithm are integrated to support navigation through the model while preserving secondary notation. We will use a distributed heating control system as an example to demonstrate our zoom and line routing algorithm.

In Fig. 9, we can see an abstracted version of the heating control system modeled in ADORA that shows the general hierarchical composition in one master module and several room modules. They are modeled as abstract object and object set represented as rectangle and pile of rectangles respectively. The three dots appended to the object names indicate that details are currently hidden. An operator can control the complete system, setting default temperatures for the rooms. Actors are represented by sexangles connected to scenariocharts. A scenariochart models an instance scenario represented by an adapted form of Jackson JSP diagrams [14, 12]. States are represented as rounded rectangles. Objects and states together form one hierarchical statechart.

Zooming into the object HeatingOn yields the situation in Fig. 10. Our zoom algorithm is applied to extend the object and adapt the surrounding context correspondingly. The inner objects and their associations to other objects are revealed. Our line routing algorithm adapts the routings of associations if required.

If we compare the two situations in Fig. 9 and Fig. 10, we can notice the secondary notation has been preserved well by our zoom and line routing algorithm. The relative positions among the objects have been kept stable. For example, RoomTempControlPanel can be found at the top right of HeatingOn before and after the zoom operation. And also the line routings have been adapted in a intuitively useful way. Both newly drawn association from the BoilerControl to the RoomControl and Settings object, for example, follow the path of the existing, hierarchical superposed association from BoilerControl to RoomModule yielding a pleasant layout.

6 Related Work

The existing approaches to visualization and editing hierarchical models are very limited concerning node positioning and employ rather primitive line routing techniques only (see Section 2). Significant work on line routing, however, has been done in domains other than modeling, where line routing is a major concern:

In the field of automatic *graph drawing* the routing of the edges that connect the nodes is an integral part of the node placement algorithm. The graphs visualize existing data and the drawing algorithm therefore can generate the

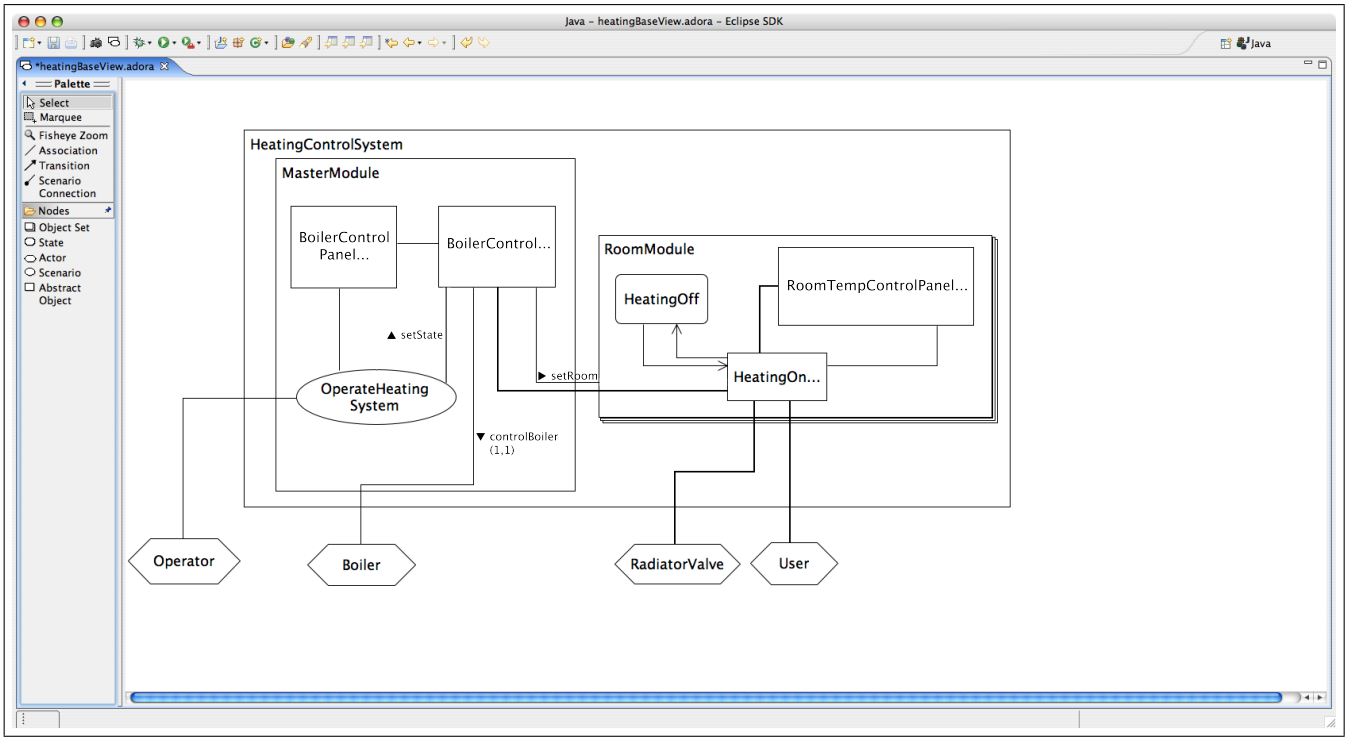

Figure 9. Screenshot of the ADORA modeling tool

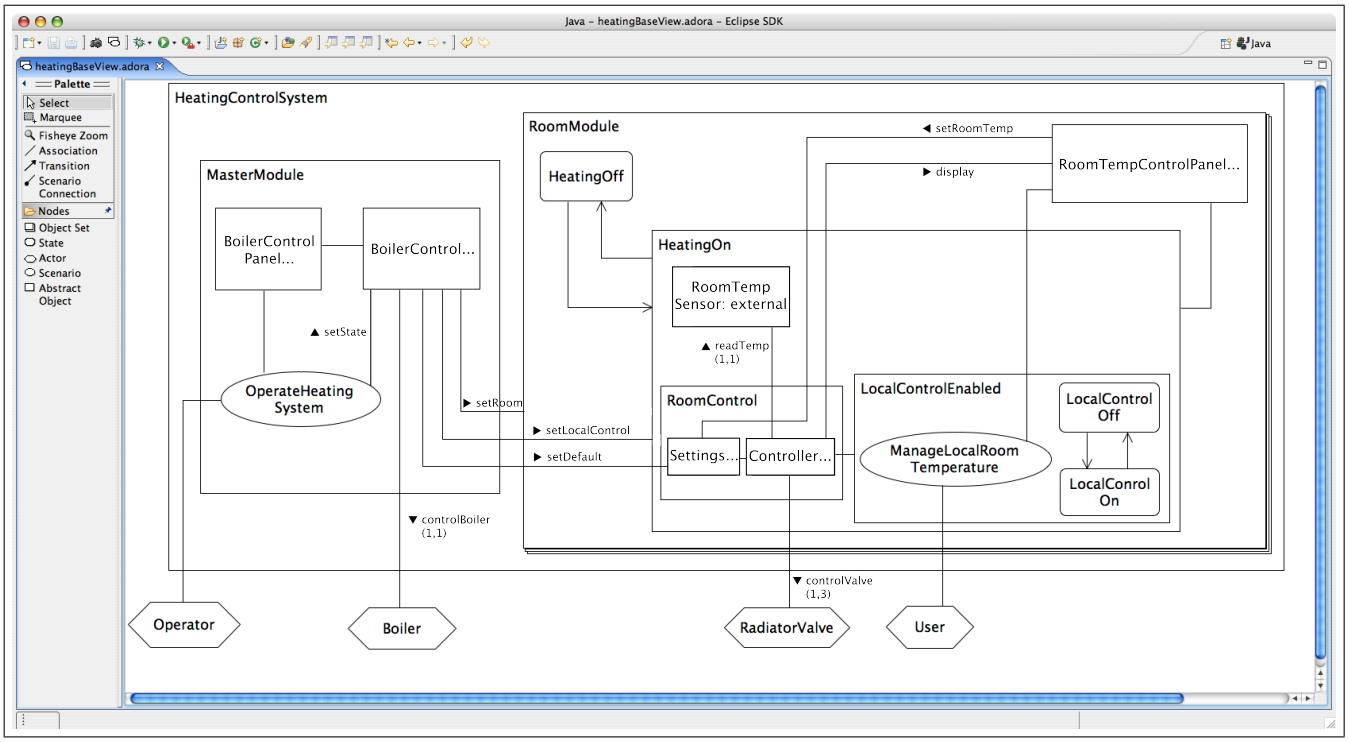

Figure 10. Screenshot of the ADORA modeling tool showing the expanded heating system

layout for all the nodes and edges together. There is no need for incremental layout adaptation or preservation of a layout produced by a human. The nodes have to be laid out in a manner that places connected nodes close to each other and avoids line intersections (see e.g. [6]).

The wire routing problem that occurs in*VLSI design* has to deal with a fixed placement of the circuit components and therefore is similar to our line routing problem. The most important routing algorithms in this domain are those developed by Lee [15] and Soukup [25] and their variants. Due to their runtime complexity, wire routing algorithms cannot be computed in real time on a personal computer.

The geometric shortest path problem in *computational geometry* has many applications in robotics, geographic information systems and diagram drawing. The best known approach constructs a visibility graph [17] and computes the shortest path on this graph according to Dijkstra's approach [4]. A visibility graph is the undirected graph that has the corners of each obstacle and the source and target point as vertices and in which two vertices are adjacent whenever they "see" each other, i.e. the nodes can be connected by a straight line without intersecting any obstacle. The avoidance of line crossings can't be integrated directly into the visibility graph and therefore has to be done in a second step after the routing.

There are many approaches dealing with the aesthetics and readability of visualized software structures or the visualization of networks. In [23], an approach for using a fisheye view in diagrams is described. This approach does not deal with hierachical decomposed nodes and it does not deal with an appealing layouting of the connections between the nodes.

The approach [5] visualizes hierarchical network structures by an approach which is related to a physical fisheye view [11] for diagrams as described in [23]. This approach is able to have several foci on a diagram and is able to preserve the secondary notation for the nodes. Nodes containing a hierarchy of sub nodes are resized to a small size if the content of them is hidden, whereas the context of the zoomed nodes is expanded realtively to the space freed. Unfortunately, this approach routes the lines between the nodes by directly drawing them, ie. without caring about the readability of the diagram, which may result in confusing overlapping between connections and nodes.

In [10], a line layouting algorithm is presented which takes care of several aestethics criteria, such as minimizing the crossing of the connections, evenly distributed nodes and connections with the same length in the diagram. However, this approach describes an automatic layouting algorithm without any influence of the user and without taking care of the secondary notation.

Storey et al. [26] and [27] uses (amongst other strategies) a fisheye view technique to visualize the structure of large software system. The usage of a fisheye preserves the seondary notation of the nodes. The hierarchically decomposed graph nodes are connected by direct connections During the zoom operations, confusing situations can result from the overlapping of connections and nodes.

In [9] an approach is described for automatically layouting UML classdiagrams for edges and nodes. The approach layouts diagrams in an appealing way according to several identified aesthetics criteria. Due to the fact that the user has no direct way to influence the way of layouting and the layout occurs fully automatic, the secondary notation for the nodes and the lines is not necessarily preserved. An approach which has quite a similar aim but which uses a different technique can be found in [13].

Discussions about diagram layout aesthetics criteria can be found amongst others in [28], [8] and [22]. Both are discussing the layout aesthetics on the basis of UML class diagrams.

7 Conclusions

Summary and state of work. In the field of human-centric, hierarchical modeling, a stable secondary notation plays an important role. However, generating views from the model to adapt the level of displayed detail, e.g. projecting and zooming, requires layout manipulations. Algorithms for both node positioning and line routing are required that (i) produce a new layout on the fly, (ii) generate appealing layouts, and (iii) preserve a human's secondary notation.

A survey of current tools revealed that these tools poorly support the visualization of hierarchical models, in particular due to poor line routing and insufficient preservation of secondary notation.

We have briefly presented our existing layout algorithm for node positioning. Based on a logical fisheye algorithm, new node layouts are computed for zooming, inserting, deleting, and filtering nodes. The algorithms takes the current layout into account and preserve it as far as possible.

As the main contribution of this paper, we have presented a novel line routing algorithm, which is based on the idea of using Lee's algorithm on a corner stitching data structure. To achieve our goals of appealing layout and preservation of secondary notation, we have adapted and extended the basic algorithm. All presented algorithms have been implemented in our ADORA modeling tool written in Java, which is able to generate views from hierarchical models.

Practical application. We have applied our algorithm to example models with encouraging results, both concerning usability and routing speed. As a next step, we are planning to do usability tests with practitioners. We are also confident that our algorithm will scale to large models because we benefit from the hierarchical model structure which decouples the number of lines to be rerouted. When the layout

of a node is changed, we have to reroute all directly embedded lines (not the ones of children nodes) plus all parent nodes up to the route node. The layout of any other node remains unchanged. So, the number of nodes in which lines need to be re-routed primarily depends on the depth of the hierarchy. A formal analysis of the performance of our algorithm will be part of our future work.

Limitations. Currently, our line routing algorithm is limited to rectilinear routing. Splines may be an alternative. However, determining intersections is more complex for splines. Label positioning is also not implemented yet.

Future work. In our ongoing research, we want to investigate to what extent nicer esthetics justify a more complex routing algorithm. As pointed out in Section 4, we want to exploit the tile structure also for a better positioning of line labels. Ongoing research will also include a formal performance analysis and further optimizations, in particular concerning the stability of generated layouts.

References

[1] D. V. Beard and J. Q. Walker II. Navigational Techniques to Improve the Display of Large 2-D Spaces. *Behaviour & Information Technology*, 9(6):451–466, 1990.

[2] S. Berner. *Modellvisualisierung f"ur die Spezifikationssprache* ADORA *[Model Visualization for the Specification Language* ADORA *(in German)]*. PhD thesis, University of Zurich, 2002.

[3] S. Berner, S. Joos, M. Glinz, and M. Arnold. A Visualization Concept for Hierarchical Object Models. In *Proceedings of the Thirteenth IEEE Conference on Automated Software Engineering (ASE '98)*, page 225, Washington, DC, USA, 1998. IEEE Computer Society.

[4] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerical Mathematics*, 1:269–271, 1959.

[5] J. Dill, L. Bartram, A. Ho, and F. Henigman. A continously variable zoom for navigating large hierarchical networks. In *Proceedings of the 1994 IEEE Conference on Systems, Man and Cybernetics*, pages 386–390, San Antonio, TX, USA, 1994. IEEE Computer Society.

[6] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[7] Eclipse Developer Community. *The Graphical Editing Framework*. Homepage. Available online at `www.eclipse.org/gef`, last visited June 8, 2006, 2006.

[8] H. Eichelberger. Aesthetics of class diagrams. In *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, page 23, Washington, DC, USA, 2002. IEEE Computer Society.

[9] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller. A topology-shape-metrics approach for the automatic layout of uml class diagrams. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 189–ff, New York, NY, USA, 2003. ACM Press.

[10] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.

[11] G. W. Furnas. Generalized Fisheye Views. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI'86)*, pages 16–23, New York, NY, USA, 1986. ACM Press.

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

[13] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. A new approach for visualizing uml class diagrams. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 179–188, New York, NY, USA, 2003. ACM Press.

[14] M. Jackson. *Principles of Program Design*. Academic Press, New York, 1975.

[15] C. Y. Lee. An Algorithm for Path Connections and its Applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, September 1961.

[16] D. Marple, M. Smulders, and H. Hegen. Tailor: a Layout System Based on Trapezoidal Corner Stitching. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(1):66–99, January 1990.

[17] N. J. Nilsson. A Mobile Automaton: An Application of Artificial Intelligence Techniques. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 509–520, 1969.

[18] OMG. *Unified Modeling Language (UML) Superstructure, 2.0* . Technical Report ptc/2003-08-02, Object Management Group, 2003.

[19] OTI Labs. *Eclipse Platform Technical Overview*. White Paper. Available online at `www.eclipse.org`, 2003.

[20] J. K. Ousterhout. Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Transactions on Computer-Aided Design CAD*, 3(1):87–100, January 1984.

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

[22] H. C. Purchase. Which Aesthetic has the Greatest Effect on Human Understanding? In *Proceedings of the 5th International Symposium on Graph Drawing*, pages 248–261, 1997.

[23] M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. In P. Bauersfeld, J. Bennett, and G. Lynch, editors, *Proceedings of the SIGCHI Conference on Human Factrs in Computing*, pages 83–91. ACM Press, Mai 1992.

[24] 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.

[25] J. Soukup. Fast maze router. In *Proceedings of the 15th conference on design automation*, pages 100–102, 1978.

[26] M.-A. Storey, F. Fracchia, and H. Mueller. Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th International Workshop on Program Comprehension (IWPC'05)*, page 17, Los Alamitos, CA, USA, 1997. IEEE Computer Society.

[27] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Muller. On integrating visualization techniques for effective software exploration. In *Proceedings of 1997 IEEE Sympsoium on Information Visualization (InfoVis '97)*, pages 38–45. IEEE Computer Society, 1997.

[28] D. Sun and K. Wong. On evaluating the layout of uml class diagrams for program comprehension. In *Proceedings of the 3rd International Workshop on Program Comprehension (IWPC'05)*, pages 317–326, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[29] Sun Microsystems. *Garbage Collector Ergonomics*. Manual. Available online at `java.sun.com/j2se/1.5.0/docs/guide/vm/gc-ergonomics.html`, last visited June 9, 2006, 2006.

[30] Sun Microsystems. *Tuning Garbage Collection with the 5.0 Java Virtual Machine*. Manual. Available online at `java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html`, last visited June 9, 2006, 2006.

A Appendix

A.1 Tool Support of Hierarchical Modeling with a Preservation of the Secondary Notation

The results of our study of the visualization features of current tools are summarized in Table 1. We concentrated mainly on UML tools and focused on the visualization features that these tools provide for hierarchically decomposable diagrams, i.e. composite structure diagrams, state diagrams and activity diagrams. Some tools do not support hierarchical decomposition in the mentioned diagrams, and some of them do not implement these diagrams at all. If these three diagram types were not available, we investigated other diagrams available, e.g. class diagrams. We use the following abbreviations for the type of diagrams: composite structure diagram (C), activity diagram (D), state diagram (S), all diagrams supported (A=CDS), object diagram / collaboration diagram (O), class diagram (K).

a. **Language Support** We started our investigation by having a look on the tools presented as UML 2.0 compliant on the official UML webpage. Unfortunately, only a few of the tools implement the full UML 2.0 standard. Some of them implement UML 1.4 standard with some elements taken from UML 2.0, indicated in row a of Tab. 1 by the label *UML1.4+*. Other tools implemented the UML 2.0 standard but with some small differences to the UML 2.0 standard. This is indicated by the label *UML 2.0-* in Tab. 1:

b. **Hierarchical Decomposition** There are different types of hierarchical decomposition and the corresponding layout techniques that can be found in the mentioned tools.

- *n/a* Means that the diagram type is not implemented
- - Indicates, that the given diagram type is implemented, but with no hierarchical decomposition.
- *1* Indicates the implementation of a hierarchical decomposition with a hierarchy of one nested level.
- *n* Indicates the implementation of a hierarchical decomposition with a hierarchy of n nested levels.
- *o* Means that no occlusion occur when inserting or moving child objects in the hierarchy.
- *F* Possibility to zoom into details of the hierarchy with a less detailed context (Fisheye view)
- *E* Means that the hierarchy can consist of sub diagrams which are displayed by explosive zoom.
- *S* Means that the hierarchy can be viewed by using scrolling to different locations in the model. This property can be combined with property E.

c. **Smart Space Management** The column c in Tab. 1 describes how far a smart space management is implemented in the corresponding tools. A smart space management helps when inserting new nodes or deleting nodes, to automatically expand or contract the canvas of the parent. The following abbreviations are used:

- - No Smart Space management
- *I* Means that smart space management is supported for inserting elements (automatically resize the parent), when the new element is inserted in the right lower corner of the parent.
- *D* Means that smart space management is supported for deleting elements (automatically resize the parent), when the element is located in the right lower corner of the parent.
- *S* Means that smart space management is given for inserting and deleting element anywhere in the parent.
- *R* Indicates smart space management when resizing a child element in the parent left right lower corner.

d. **Model Projection** The column d in Tab. 1 describes how elements of the model can be filtered out.

- *m* Manual projection, i.e. creating a new diagram manually by copying references of the elements that should be shown.
- *I* Automated projection facility, i.e. model parts can be selected for automatically filtering them out.
- - No projection facility

e. **Line routing** We had a look on different properties in line routing (see column d in Tab. 1). If no hierarchical model was available, we had a look on the line routing in other diagram types of the language.

- *d* Direct (straight) line routing
- *s* Splines Routing
- *m* Rectilinear line routing
- *a* The line route is computed automatically.
- *b* There is the possibility to insert bend points
- *C* Collision free line routing (no node collisions)
- *L* Overlapping free line routing
- *K* Crossing free line routing
- *M* midpoint label positioning with occlusion
- *O* Occlusion free label positioning

A.2 Performance Analysis of the Algorithm

We have analyzed the performance of our line routing algorithm by executing and measuring nine test cases. The test cases consist of ADORA diagrams with different layouting complexity.

A.2.1 A Short Overview of the Routing Algorithm's Implementation

We implemented our line routing algorithm in an Eclipse plugin [19] for the ADORA [12] language. The plugin uses the Graphical Editing Framework (GEF) [7] which provides the ability to use an arbitrary implementation of a line router. We used this line router interface of the GEF to implement our line routing algorithm. This implementation for the ADORA editor executes the following steps each time a line is routed:

Table 1. Feature matrix of investigated tools

Vendor	Tool	Version	a	b	c	d	e
Altova	Altova UModel	2005 sp1	UML1.4+	A:n/a	A: -	A:m	A:mabML
Sparx Systems	Enterprise Architect	5.00.769	UML2.0	D:nES, C:nS, S:nES	A: –	A:m	A:dM
ARTiSAN	Real-time Studio	5.0.22	UML 1.4+	D:-, S:nS, C:n/a	S:D,I	A:m	S,D:dMb
Embarcadero Technologies, Inc.	DescribeEnterprise	6.1.7.1119hEE	UML1.4+	D:-, S:nS, C:n/a	A:–	A:m	A:dMB | K:mabM
No Magic, Inc.	MagicDraw UML	9.5	UML1.4+	D:–, O:Sn, S:Sn	S,O: I	A:m	A:maMbL
I-Logix Inc.	Rhapsody	6.0 Developer	UML1.4+	D:n/a, O:I,S:Sn	A: –	A:m	S:sa, O:mabM
MathWorks	Stateflow	6.0	Other	S:Sn	S:–	S:–	S:sam
IBM Corp.	Rational Modeler	6.0.0	UML2.0–	D:– C:–, S:Sn	S: I, D	A:m	A:dbM
Borland	Together Designer	2005 (5552.0)	UML2.0–	D:–, S:Sn, C:S1	S: I, D, C:–	A:m	A:dbMO
Visual Paradigm	UML Enterprise Edition	5.0	UML2.0–	C:Sn, S:SEn	A:-	A:m	A:dbaM
Gentleware	Poseidon Community Edition	3.1.0 CE	UML1.4+	D:–, S:SEn,	S: I, D	A:m	AdbM
University of Paderborn	Fujaba	4.0.1	UML1.4	D:–, S:nS, C:n/a	S: I,R	A:–	S:mbaM

Table 2. Performance Test Configuration Line Routing

Hardware/Software	Speed/Size/Version
Pentium 4	3 Ghz
RAM Memory	2 GB
Windows XP Professional	Version 2002 - Service Pack 2
Sun Java Runtime Environment	1.5.0_06-b05, mixed mode
Eclipse Software Development Kit	3.1.2 - M18012006-1600
Graphical Editing Framework	3.1.1
Adora Editor Plugin	Version pre-1.0.0, build 186

1. The start and the end point of the line are determined. The start and the end point are represented by the click points, i.e. the start and the end point where the user clicked to draw the line.

2. The tile structure is created. This is currently done each time a line is routed. However, a much more efficient way would be just to do it the first time in a series of lines that have to be routed.

3. The white space tiles are determined. This is done according to the algorithm described in 4.1.2.

4. The exact path of the drawn rectangular line segments which connect the start and the end tile are calculated.

5. The connection is drawn.

A.2.2 The Test Configuration and the Performance Test Cases

The test environment used for the performance test consisted of the configuration shown in Tab. 2. The line routing algorithm is contained in the implementation of the ADORA plugin. Fig. 11 – Fig. 19 show the diagrams which were used as performance test cases. Some of the diagrams were prepared in a way that their routing computation was more complex. We achieved this by positioning the nodes and by setting the start and end point of the lines so that more obstacles between the start and the end tile resulted and therefore the computing effort was bigger.

The described positioning of the nodes and the click points results in a rather unappealing layout which is the case for the shown performance test cases. Additionally, normal ADORA diagrams should not show too many model elements at the same time. The user of the tool would hide several elements, e.g. connections, in the model to reduce the cognitive overhead as described in [2].

A.2.3 The Performance Test Results

For the test cases shown in Fig. 11 – Fig. 19, we measured the time for the routing of each line in the given test cases. Concretely, the value of the system clock immediately before and after the execution of the routing algorithm for each single line was determined. The results of this test are given in Tab. 3 to 8. When analyzing these results, one has to take into accout that several factors may bias the measured time.

Firstly, taking the starting time as well as the ending time of routing algorithm takes time itself. However, the computation time required for taking the current time can be neglected as it is rather small and contained in all samples.

Secondly, the implementation of the routing algorithm is written in Java and was executed with a standard JVM of Sun's Java Runtime Environment which uses a state of the art garbage collector. This garbage collector is executed

non-deterministically. When the garbage collector is run, it uses a non-deterministic period of time which is bounded by an upper limit [29, 30]. For executing the test, we used the default settings for the garbage collector of the JVM. Hence, particular results were biased by the execution of the garbage collector. Some of these results may be:

- Test case 6 (Tab. 5)
  - Connection 15/16
  - Connection 18/17
- Test case 9 (Tab. 8)
  - Connection 15/16
  - Connection 27/23
  - Connection 36/29

These connections have a significantly higher time needed for computing the route of the connection and therefore its highly probable that during the calculation of these connections, the garbage collector was executed.[3]

In our test sample, the average time required for routing one connection is 3.33 milliseconds. Assuming that a user is irritated when having a system reaction time greater than 500 milliseconds, about 151 lines can be routed in sequence without user irritation. This value represents a lower bound for the actual performance of the algorithm. A better performing implementation (for example, one which computes the tile structure only when necessary and blocks garbage collection while computing a diagram) will result in a much higher number of lines that can be routed in less than 500 milliseconds. Moreover, the current implementation of the above algorithm in GEF 3.1.1 calculates and draws the connection routes asynchronously which mitigates the problem of an irritated user dramatically. Additionally, one should take into account that a diagram with 151 connections is rather unreadable. Most of the connections in such a case are not in the focus of interest and may be hidden for the sake of understandability. A view mechanism capable of doing this is described in [12]. Hiding connections that are not in the focus of interest additionally helps reduce the computational effort for line routing.

In summary, our analysis demonstrates that the algorithm described in section 4 is suitable for real time routing of connections in hierarchically decomposed diagrams.

Table 3. Performance test results for the test cases 1– 4 shown in Fig. 11 – Fig. 14

Test Case No.	Start Node	End Node	Time in Milliseconds
1	2	1	0.17
2	2	1	0.39
3	2	1	1.06
4	2	1	1.18
	3	4	1.20
	5	6	0.82

Table 4. Performance test results for the test case 5 shown in Fig. 15

Test Case No.	Start Node	End Node	Time in Milliseconds
5	2	1	0.37
	4	3	0.41
	6	5	0.64
	7	8	1.48
	9	9	2.48

[3] Actually, we did not check if and when the garbage collector was running during the test execution.

Table 5. Performance test results for the test case 6 shown in Fig. 16

Test Case No.	Start Node	End Node	Time in Milliseconds
6	2	1	0.37
	4	3	0.47
	6	5	0.64
	7	8	1.07
	9	10	2.12
	11	12	4.88
	13	14	9.04
	15	16	16.91
	18	17	25.84

Table 6. Performance test results for the test case 7 shown in Fig. 17

Test Case No.	Start Node	End Node	Time in Milliseconds
7	2	1	0.60
	4	3	2.13
	6	5	5.81
	7	8	4.30
	9	10	4.75
	11	12	1.05
	13	14	1.04
	15	16	0.83
	18	17	0.62

Table 7. Performance test results for the test case 8 shown in Fig. 18

Test Case No.	Start Node	End Node	Time in Milliseconds
8	2	1	1.26
	4	3	0.60
	6	5	0.51
	7	8	0.87
	9	10	0.80
	11	12	1.35
	13	14	0.91
	15	16	1.21
	18	17	0.60

Table 8. Performance test results for the test case 9 shown in Fig. 19

Test Case No.	Start Node	End Node	Time in Milliseconds
9	2	1	2.31
	4	3	0.92
	6	5	1.08
	7	8	1.86
	9	10	3.77
	11	12	6.35
	13	24	6.12
	15	16	10.88
	32	17	7.36
	18	17	8.64
	28	19	0.85
	21	34	3.53
	23	25	2.60
	26	23	6.91
	27	23	12.12
	25	36	8.44
	36	29	14.38
	20	32	0.81

Figure 11. Performance test case 1– A simple connection routed from node 1 to node 2.

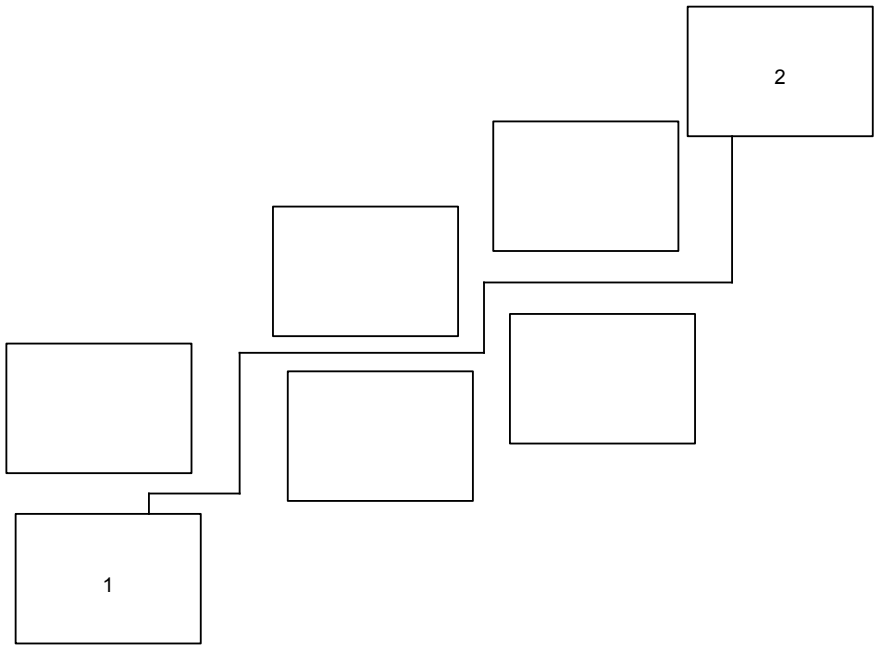

Figure 12. Performance test case 2 – A connection between node 1 and 2 routed through several obstacle nodes

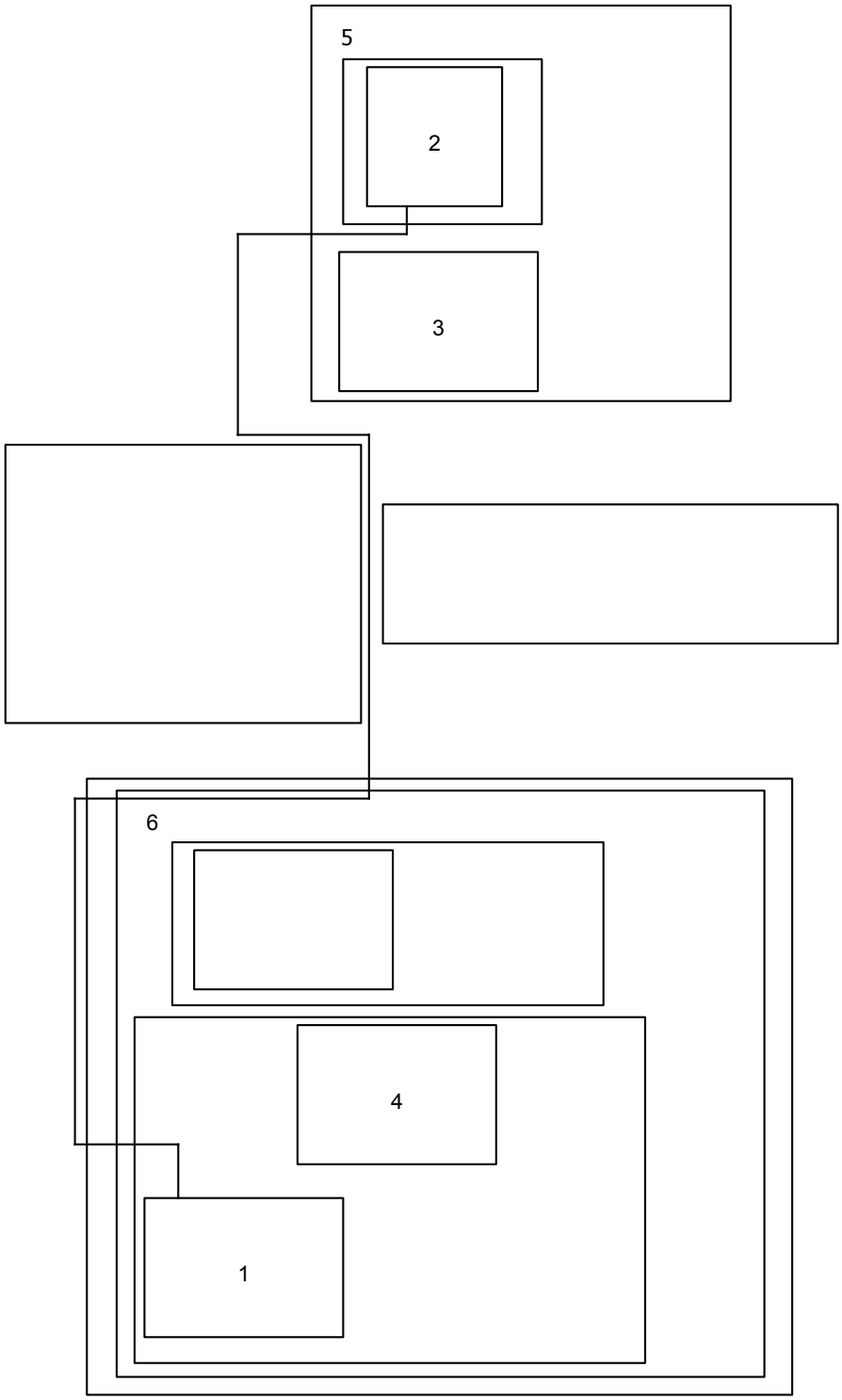

Figure 13. Performance test case 3 – A connection routed from node 1 to node 2 in a hierarchie of nested nodes.

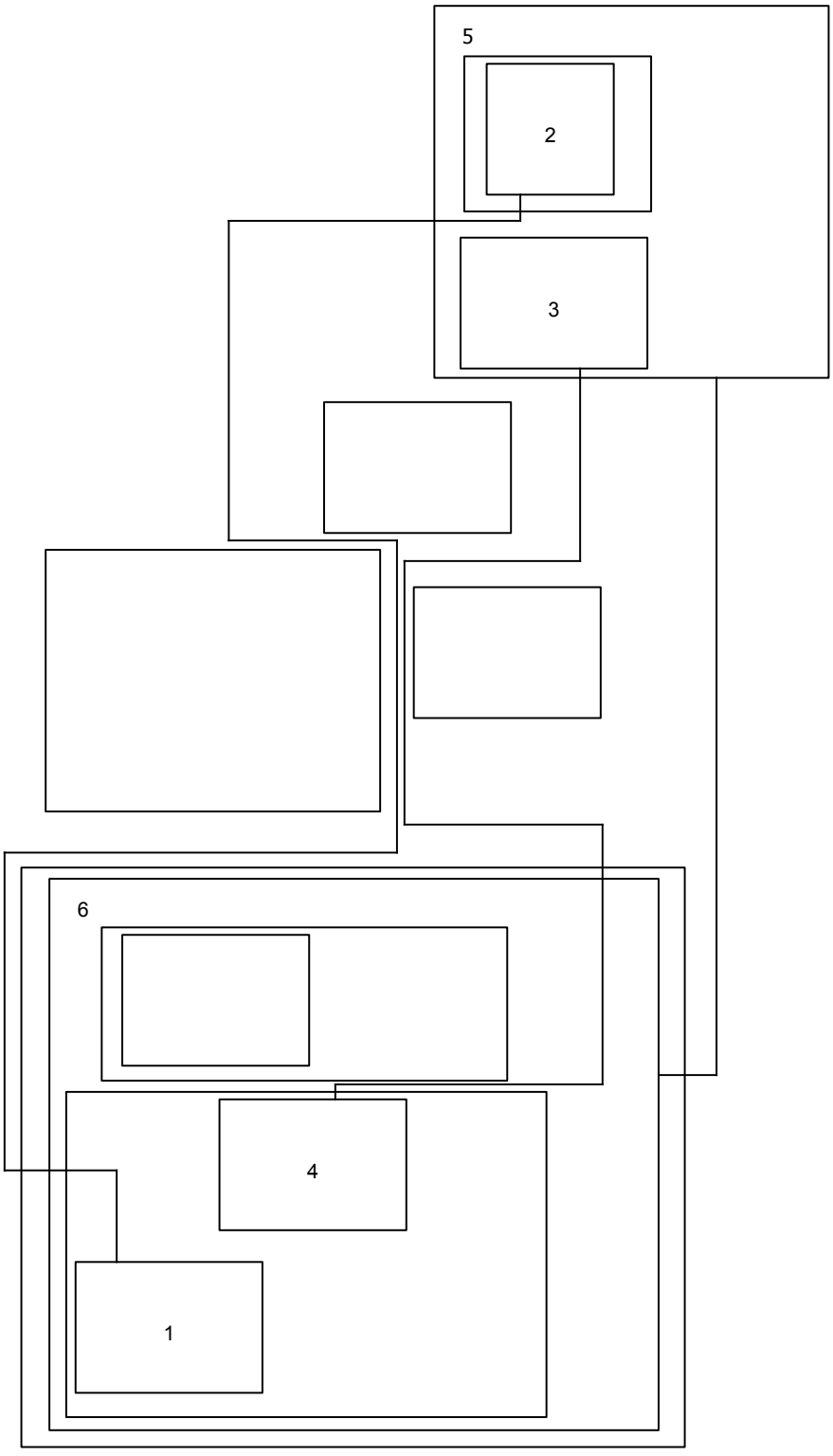

Figure 14. Performance test case 4 – Routing a connection from node 1 to node 2 and another from node 3 to node 4 in a diagram with nested nodes.

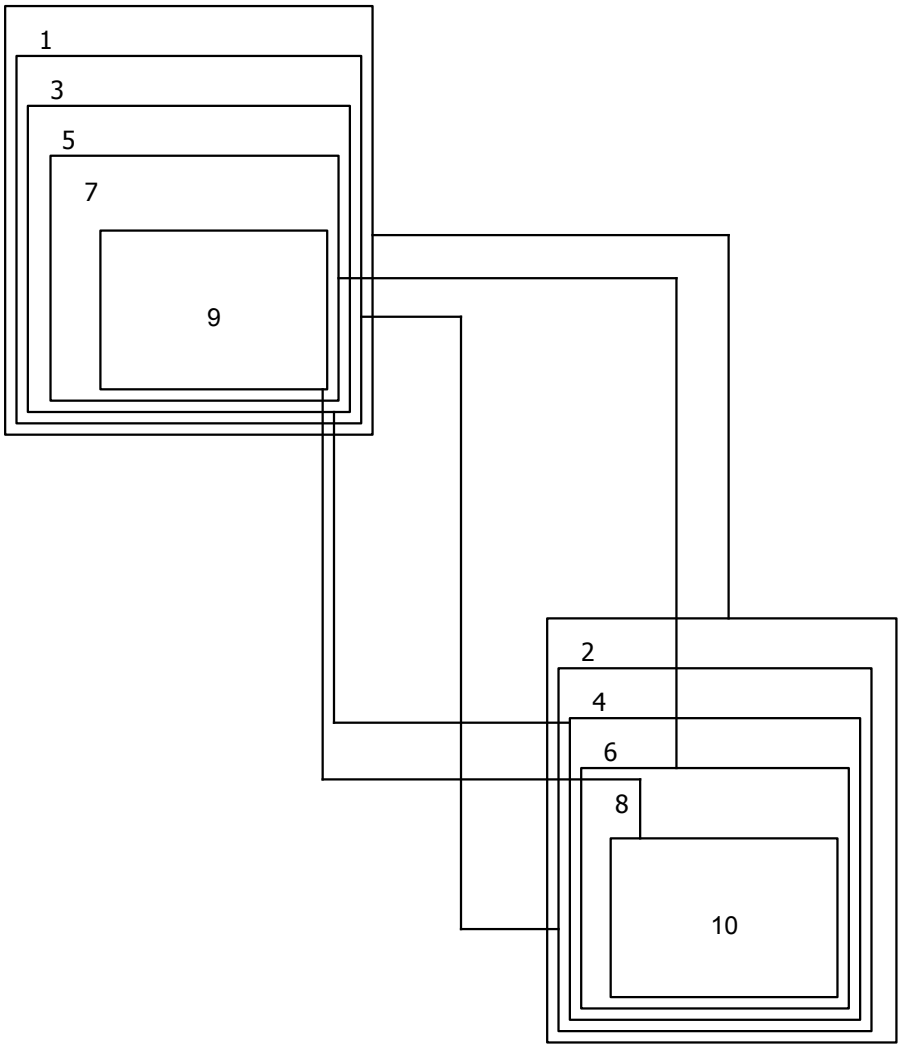

Figure 15. Performance test case 5 – Routing several connections in a diagram of deeply nested nodes.

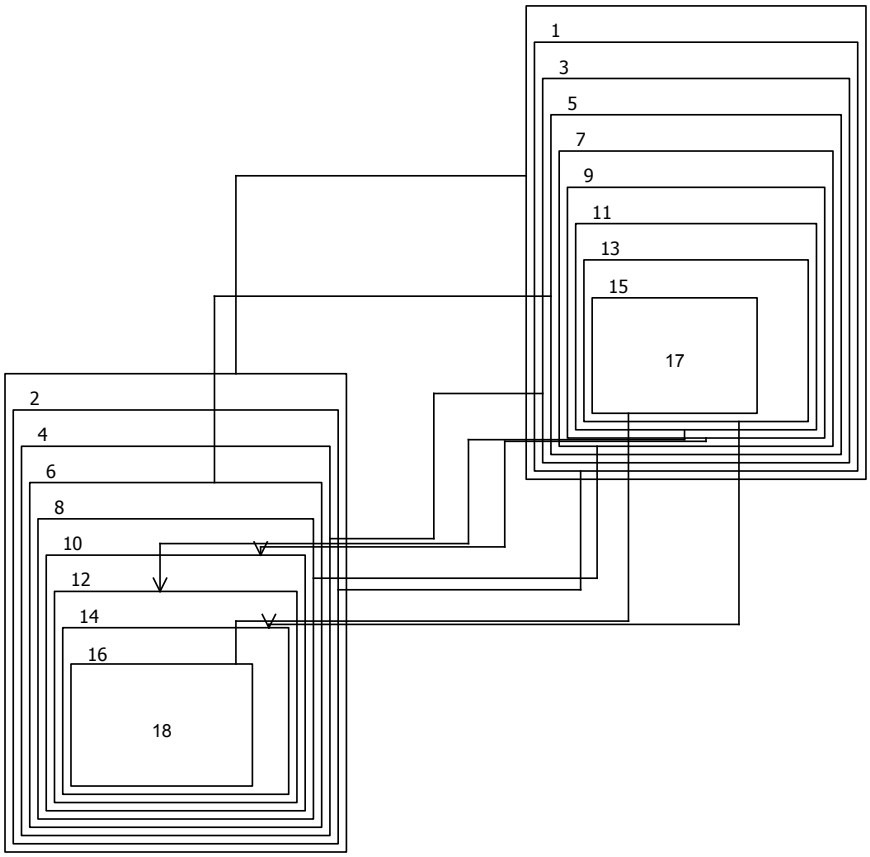

Figure 16. Performance test case 6 – Another routing of several connections in a diagram of deeply nested nodes.

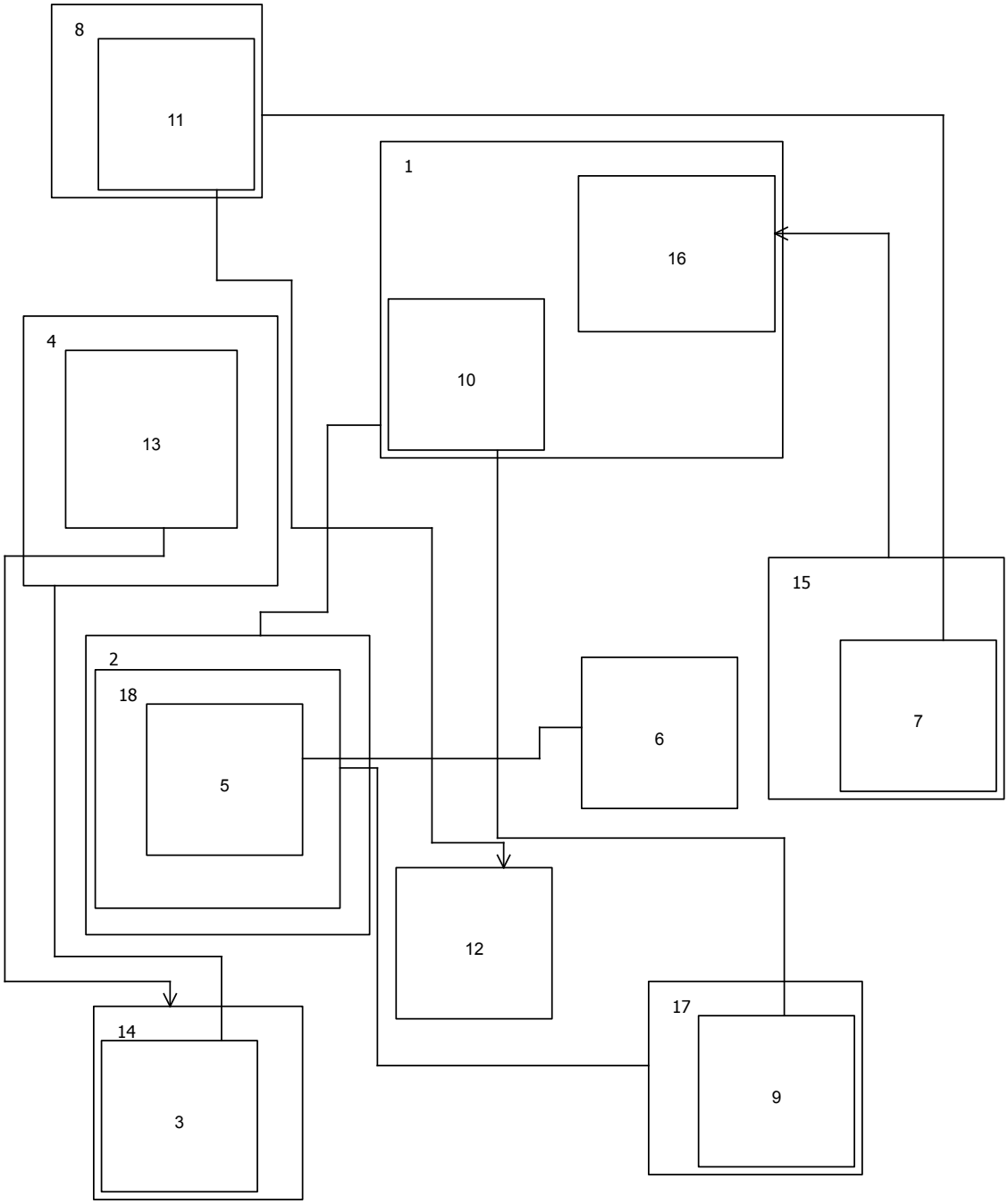

Figure 17. Performance test case 7 – A more complex diagram with several nested nodes and several connections.

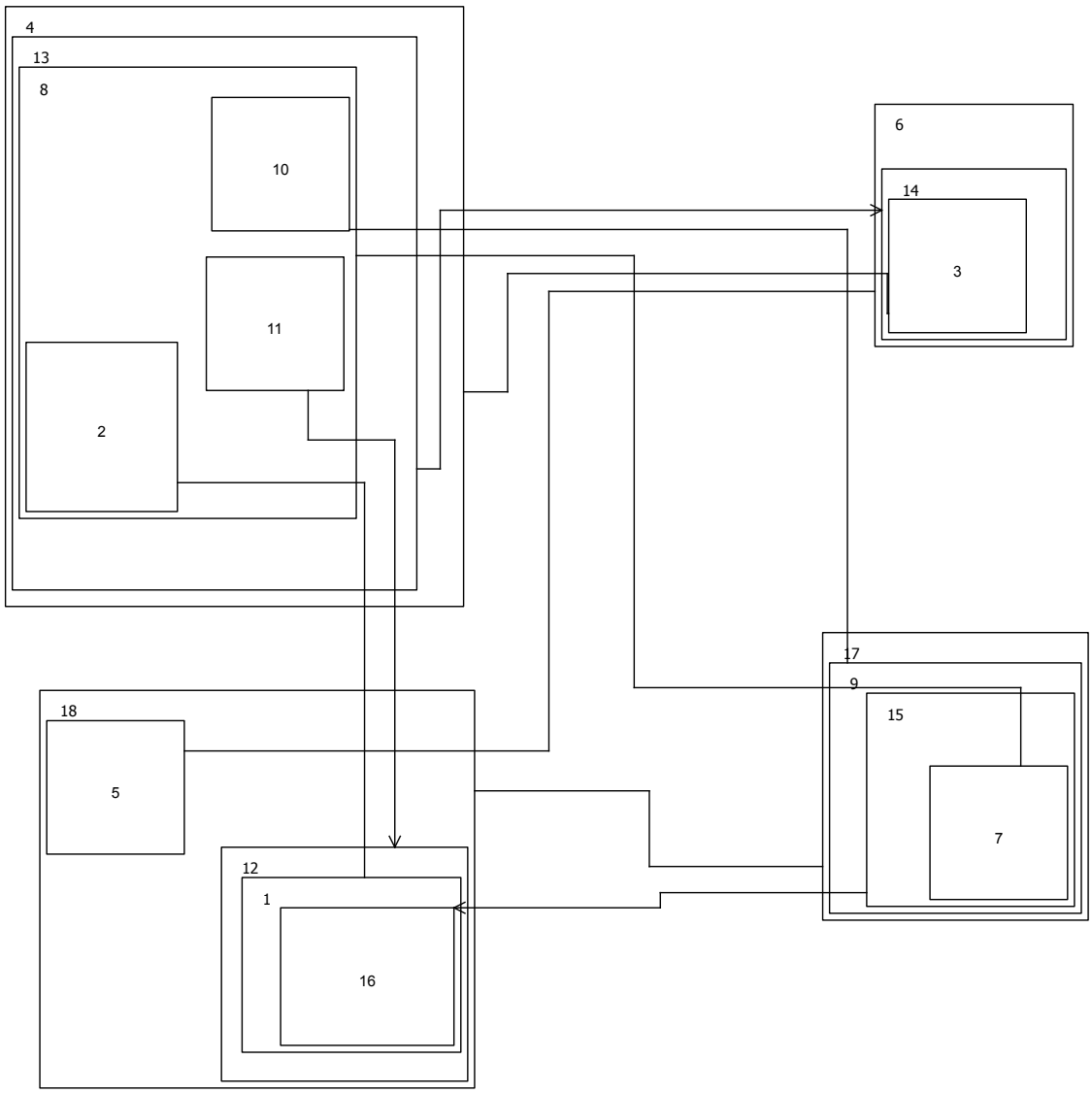

Figure 18. Performance test case 8 – A more complex diagram with several nested nodes and several connections.

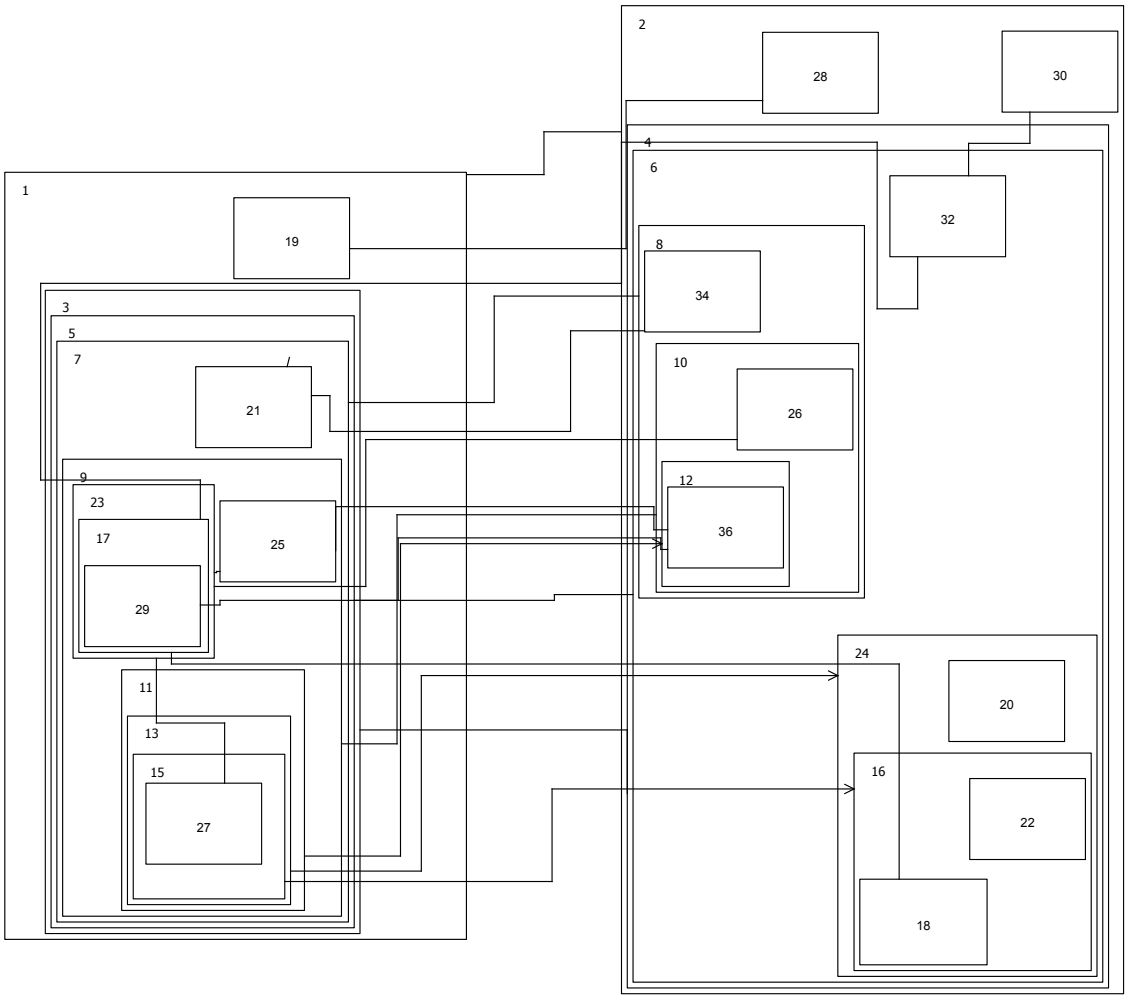

Figure 19. Performance test case 9 – Another complex diagram with deeply nested nodes and several connections.