Textual Diagram Layout Language and Visualization Algorithm

Balázs Gregorics, Tibor Gregorics, Gábor Ferenc Kovács, András Dobreff, Gergely Dévai
Eötvös Loránd University, Faculty of Informatics
Budapest, Hungary
{grbtaai,gt,koguaai,doauaai,deva}@inf.elte.hu

***Abstract*—Graphical diagrams are an excellent source of information for understanding models. On the other hand, editing, storing and versioning models are more efficient in textual representations. In order to combine the advantages of these two representations, diagrams have to be generated from models defined in text. The generated diagrams are usually created by autolayout algorithms based on heuristics.**

In this paper we argue that automatically laid out diagrams are not ideal. Instead, we propose a textual layout description language that allows users to define the arrangement of those diagram elements they consider important. The paper also presents algorithms that create diagrams according to the layout description and arrange the underspecified elements automatically.

The paper reports on the implementation of the proposed layout description language as an embedded language in Java. It is used to generate class and state machine diagrams compatible with the Papyrus UML editor.

I. Introduction

Textual modeling has many advantages compared to graphical one: It is faster for experienced developers to edit models in text rather than to edit graphics, which is partly the consequence of the maturity of text editors compared to graphical model editors. Model storage and version control is also easier for plain text. On the other hand, graphical visualization of certain kinds of models, like UML for example, is essential: Understanding a model is much easier by looking at an expressive diagram than reading text.

This duality lead to research for text-diagram coexistence: The resulting tools either convert the different representations back and forth or they use projectional editing [17]: special text and graphical editors are views of a common representation in the background. In both cases, the layout of diagrams is a problem to solve. Since textual representations usually contain no information about layout, it has to be inferred automatically by algorithms using smart heuristics.

However, we argue that automatically laid out diagrams are not ideal, because the layout of hand-crafted diagrams contain extra information not present in the models themselves: Most important concepts are usually centered in the diagram, while technical entities are placed aside. Links to be emphasized are short and straight, while less important ones might go around the diagram just not to cross the substantial ones. Another problem is that a single new model element can completely change the automatically computed layout of the whole diagram, making orientation hard for the users. As a response to these problems, layout algorithms have been constructed that preserve the location of elements manually (re)placed by the user or fixed by a previous run of the algorithm.

Considering that layout information is something important to store together with the model and that storage and version control is much easier for plain text than graphics, we argue that having textual modeling languages is not enough: There is also need for *textual diagram layout languages*. Such a language should provide powerful constructs to enable concise layout descriptions. It should be possible to partially set the layout, concentrating on the elements that the user considers relevant, and have the underlying algorithms arrange the rest automatically.

This paper presents a layout description language designed for *txtUML* [5], a domain specific language for executable UML modeling, embedded in Java. For this reason, the implementation of the presented layout description language is also embedded in Java, but the concepts and algorithms are independent of this particular implementation technique.

The main contributions of the paper are the following:

- A domain specific language for diagram layout description. It can be applied to any kind of diagrams consisting of boxes and arrows, for example UML class diagrams or state charts. The current version is applicable for flat diagrams. Support for boxes contained in other boxes is future work.
- Algorithms to generate diagrams according to the layout descriptions. The proposed algorithms can decide if it is possible to satisfy the constraints implemented in the layout description. If the description does not fix all elements, it also takes into account general recommendations about the arrangement of UML diagrams.
- The language and the algorithms are implemented as part of the *txtUML* modeling language and can generate Papyrus [1] diagrams.

The next section describes related work, then section III presents the language frontend. The frontend constructs are translated to constraints of a core language, presented in section IV. The algorithms, that lay out first the boxes and then the links according to the constraints, are described in sections V-B and V-C. Section VI gives an overview of our implementation, then the proposed solution is evaluated and summarized in sections VII and VIII respectively.

II. Related Work

Ti*k*Z [16] is a LaTeX package for creating vector graphics. Its basic concepts include nodes and edges, similarly to diagrams used for modeling. The relevant part of Ti*k*Z from our perspective is how the placement of nodes and edges are defined. The basic solution is using coordinates, which is clearly necessary to define vector graphics precisely, but would be really inconvenient for a diagram layout language aiming for concise and readable notation. More interesting is that the place of nodes can be defined relatively to each other, while for edges it is possible to define at which side of the corresponding nodes they start and end. For example, node a is right of b and a link from a to b exits at the west side of a and enters at the east side of b. These concepts are also part of our language defined in section III.

The TikZ-UML [10] package is built on top of Ti*k*Z and supports the most common UML diagrams. Boxes of the diagrams are positioned using plain coordinates, but the package includes interesting constructs to define the geometry of lines: For example, $-|-$ means that the line should have horizontal-vertical-horizontal geometry, which can be fine-tuned with coordinates. Anchor points of lines can be specified by angular values in degree, 0 meaning east, 90 north etc. The problem with this approach is that a given angular value can fall on different sides of the box depending on its dimensions. For this reason, in our language the side of the box can be defined explicitly and we let the algorithm determine the exact anchor point automatically.

MetaUML [7] is a MetaPost library for typesetting UML diagrams. Arrangement of the diagram elements is done by explicitly setting coordinates, but syntactic sugar is present in the language to describe common cases easier: There are commands to create rows or columns of elements with a given uniform spacing between them. Other statements express that two elements are on the same horizontal or vertical line and further statements are used to define spacing. Similar statements are also present in our layout language, but spacing is always computed implicitly by placing the boxes on a grid. Similarly to Ti*k*Z, it is possible to define at which sides does a link touch the connected nodes. Arbitrary paths can be drawn by providing exact coordinates of midpoints and specifying if curves or lines should be used for connections. There is special support for *Manhattan path* that consist of one vertical and one horizontal segment and also for *stair step path* that include two vertical and one horizontal segments (or vice versa) and form a stair step. Our algorithm in section V-C often produces Manhattan and stair step paths, because it uses vertical and horizontal segments and searches for the shortest path according to a cost function to be discussed in that section.

Layout managers for programming graphical user interfaces also solve the problem of defining graphical layout in text. They provide abstractions to make layout definitions easier to write and maintain. For example, Java AWT and Swing layout managers [13] use concepts like

- placing elements to top, bottom, left or right (BorderLayout),
- creating rows or columns of elements (BoxLayout),
- arranging elements according to a grid (GridLayout).

The abstractions of the first two items are explicitly present in the language we created (section III), while a grid is used by our visualization algorithm (section V).

It is important to stress that we are solving a problem significantly different from that of autolayout algorithms. Those apply built-in heuristics to achieve aesthetic diagrams [14], for example to minimize link crossings and to arrange classes in a tree according to generalization relations. Our primary goal, in contrast, is to create a diagram according to the (partial) layout definition provided as input. Part of the task is to decide if it is possible to realize the layout description or it is contradictory. The layout definition may also give some freedom in placing certain diagram elements. In that case our algorithm also tries to find a solution according to the heuristics used by autolayout algorithms. However, this is not of primary importance in our case: If the resulting diagram is unsatisfactory, the user can easily fix that by extending the layout definition.

Section V-B will describe that our layout descriptions are translated to constraints that are solved in order to get the diagram. Similar constraints are used in [3], but they integrate the constraints into the Sugiyama algorithm [15] which is for visualizing hierarchies rather than general diagrams. In [9] more complex constraints are considered than the ones our language requires. The paper [11] presents interactive user - machine optimization of graph layout, using simple horizontal and vertical alignment constraints as one of the ways the user can influence the layout.

To the best of our knowledge, our approach uniquely combines textual descriptions, layout constraints and autolayout heuristics. While the goal of conventional autolayout algorithms is to create aesthetical diagrams, our goal is to allow modelers to easily create, store and version control any layout they wish. On the other hand, the textual layout descriptions mentioned in this section require full definition of the layout, while our method can handle partial layout descriptions.

III. Layout Language

The language provides constructs to create diagrams, to specify their contents and to define their layout. Currently we support flat diagrams consisting of boxes and links between them, like class diagrams or state charts without hierarchical states. We have defined language constructs to express relative positioning of boxes and to describe the geometry of link-box connections. It is also possible to form groups of boxes and define the relative positioning of the groups.

The meaning of the layout statements in this section is described informally. Their formal semantics is given in section IV by defining constraints on the coordinates of the boxes for a subset of the statements (called the core language) and by translating all other statements to core language ones.

A. Elements of the Diagram

The diagram will contain all boxes that are involved in at least one layout statement or group. Regarding links, all links that connect boxes on the diagram will be shown even without appearing in the layout description.

There might be boxes that the user wants to have on the diagram without any specific layout constraints. These boxes can be added using the $Show$ statement:

- $Show(ExampleBox)$ where *ExampleBox* is a class or state defined in the model.

$Show$ can also be used for links and can accept multiple parameters for convenience:

- $Show(ExampleLink)$ adds the link *ExampleLink* and the boxes it connects to the diagram.
- $Show(ExampleBox, ExampleLink, \dots)$ adds all enumerated boxes and links (and boxes connected by the links) to the diagram.

The user may create boxes just for positioning other boxes, this is done by using the $Phantom$ statement:

- $Phantom(ExamplePhantom)$ where *ExamplePhantom* is a unique box name. This statement creates a virtual box that the user can refer to in other statements. Phantom boxes do not appear on the diagram.

B. Box Positioning

The simplest constructs describe the positioning of two neighbors relative to each other:

- $Above(Box_1, Box_2)$ means that Box_1 is above Box_2 and they are neighbors.
- The statements $Below$, $Left$ and $Right$ can be used similarly.

The next set of constructs is similar, but they imply weaker constraints:

- $North(Box_1, Box_2)$ means that Box_1 is north of Box_2 and their horizontal positions are unconstrained. That is, both their vertical and horizontal distance can be any large, the only constraint is that the diagram can be split into two parts by a horizontal line such that Box_1 is in the upper and Box_2 is in the lower part.
- The statements $South$, $West$ and $East$ are similar.

The following language elements define frequently used layout patterns:

- $Row(Box_1, \dots, Box_n)$ creates a row of neighboring boxes, that are on the same horizontal line, left to right in the order of their appearance in the statement.
- $Column(Box_1, \dots, Box_n)$ creates a column of neighboring boxes, that are on the same vertical line, top down in the order of their appearance in the statement.
- $Diamond(top = Box_1,\ left = Box_2,\ right = Box_3,\ bottom = Box_4)$ arranges the four boxes in a diamond shape: An empty place in the middle surrounded by Box_1, Box_2, Box_3 and Box_4 as neighbors at the top, left, right and bottom respectively.

Selected boxes can be positioned relative to all other boxes using the following statements:

- $TopMost(Box_1, \dots, Box_n)$ means that the given boxes are north of all other boxes and their horizontal positions are unconstrained. That is, the diagram can be split into two parts by a horizontal line such that the given boxes are in the upper and all the rest is in the lower part.
- The statements $BottomMost$, $LeftMost$ and $RightMost$ are similar.

C. Positioning of Links

The following statement can be used to set how important selected links are:

- $Priority(ExampleLink, 10)$ assigns the priority 10 to the link $ExampleLink$. Links with higher priorities will be laid out earlier to make sure they become as short as possible and turn only if inevitable.

It is possible to define which side of a box should a link connect to:

- $North(ExampleLink, ExampleBox)$ specifies that the link $ExampleLink$ should be connected to box $ExampleBox$ from the north.
- $North(ExampleLink, ExampleBox, Start)$ is used in case of reflexive links, the last parameter can be used to indicate which end of the link the statement specifies the direction for ($Start$ or End).
- The statements $South$, $West$ and $East$ can be used similarly.

D. Groups of Boxes

Boxes can be grouped together for two reasons: First, it is possible to define the relative positions of the members in the group, which might be more convenient than doing it pairwise. Second, it is possible to set the relative position of groups or a group and a box.

- $Group(ExampleGroup, Box_1, \dots, Box_n)$ means that the boxes $Box_1, \dots, Box_n$ will be part of the group $ExampleGroup$.

The contents of the group can be aligned in the order they were put into it.

- $Alignment(ExampleGroup, TopToBottom)$ means that the contents of group $ExampleGroup$ are aligned top to bottom. If $ExampleGroup$ contains the elements $Box_1, \dots, Box_n$ then Box_2 is south of Box_1, Box_3 is south of Box_2 and so on.
- The alignments $BottomToTop$, $LeftToRight$ and $RigthToLeft$ can be used similarly.

A group can be used in any statement where a set of boxes can be used. For example, instead of $Row(Box_1, \dots, Box_n)$ the user can use $Row(ExampleGroup)$ if the group $ExampleGroup$ is defined somewhere.

The statements $North$, $South$, $West$ and $East$ accept groups as either of their parameters. It is also possible to use an "unnamed group" as one of the parameters:

- $North(ExampleGroup, ExampleBox)$ specifies that all elements of $ExampleGroup$ has to be placed north of $ExampleBox$.
- $West(ExampleGroup, \{Box_1, \ldots, Box_n\})$ means that any element of $ExampleGroup$ has to be west of Box_i for any $i \in [1..n]$.

E. Example

The following example shows a layout definition for a class diagram. The model contains the $Paper$ class, instances of which are written by $Author$s and reviewed by $Reviewer$s. $CoAuthor$ is a reflexive association between authors. Papers are published at $Conference$s, which are specialized to $MainConference$s or $Workshop$s.

- $Row(Author, Paper, Reviewer)$
- $Above(Paper, Conference)$
- $Show(MainConference, Workshop)$
- $West(CoAuthor, Author, Start)$
- $South(CoAuthor, Author, End)$

The Row statement arranges the classes $Author$, $Paper$ and $Reviewer$ to a horizontal line in this order, left to right. The $Above$ statement specifies the location of $Conference$ relative to $Paper$. Classes $MainConference$ and $Workshop$ are added simply using the $Show$ statement. There is no need to add explicit constraints on these classes, because the algorithm presented in section V-B adds implicit layout constraints for generalization relations.

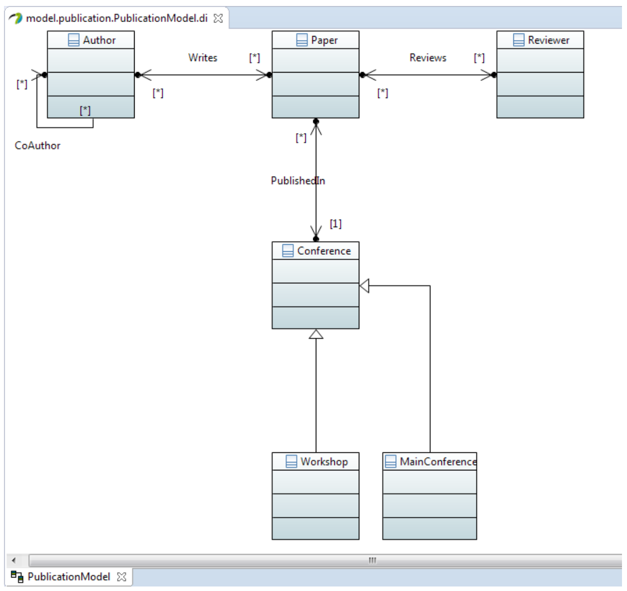

Fig. 1. Generated Papyrus diagram

Figure 1 shows the Papyrus diagram automatically generated from the model and the layout definition above.

IV. Core Language

The core language is a proper subset of the previously described layout language. Its aim is to express the layout language with fewer and simpler linguistic elements.

The following statements express constraints between two boxes: $north(A, B)$, $east(A, B)$, $south(A, B)$, $west(A, B)$, $above(A, B)$, $right(A, B)$, $below(A, B)$, $left(A, B)$. In order to simplify the definition of the formal semantics of these statements, we treat the boxes as points on a grid. The distance of two neighboring grid points is defined to be 1. Using this, the above statements can be defined as below:

- $north(A, B)$: $A.Y > B.Y$
- $east(A, B)$: $A.X > B.X$
- $south(A, B)$: $A.Y < B.Y$
- $west(A, B)$: $A.X < B.X$
- $above(A, B)$: $A.Y = B.Y + 1$, $A.X = B.X$
- $right(A, B)$: $A.Y = B.Y$, $A.X = B.X + 1$
- $below(A, B)$: $A.Y = B.Y - 1$, $A.X = B.X$
- $left(A, B)$: $A.Y = B.Y$, $A.X = B.X - 1$

Constraints on the connection of links to boxes are given by the following statements. They specify which side of the box (B) is the link (L) connected to: $north(L, B)$, $east(L, B)$, $south(L, B)$ and $west(L, B)$. In case of reflexive links, we also have to define which end of the link we want to give a constraint on: $north(L, B, Start)$, $north(L, B, End)$, $east(L, B, Start)$ etc.

In order to define the semantics of these, we need boxes with dimensions. Let d denote the width and height of a square shaped box B, $(B.X, B.Y)$ its top-left corner, and $(L.X, L.Y)$ the appropriate endpoint of a link L at the box. The semantics of the above statements are as follows:

- $north(L, B) : B.Y = L.Y, B.X < L.X < B.X + d$
- $east(L, B) : B.X + d = L.X, B.Y - d < L.Y < B.Y$
- $south(L, B) : B.Y - d = L.Y, B.X < L.X < B.X + d$
- $west(L, B) : B.X = L.X, B.Y - d < L.Y < B.Y$

There are three technical core language statements: $show(P)$, $phantom(P)$ and $priority(L, Value)$, which add (visible and invisible) elements to the diagram and affect the order in which links are layed out. These statements have no formal semantics in terms of coordinates.

The rest of the language can be translated to core language elements as shown below:

- $Row(B_1, \ldots, B_n)$:
  $\forall i \in [1..n-1] : left(B_i, B_{i+1})$
- $Column(B_1, \ldots, B_n)$:
  $\forall i \in [1..n-1] : above(B_i, B_{i+1})$
- $Diamond(top = A, left = B,$
  $bottom = C, right = D)$:
  $phantom(P), above(A, P), left(B, P),$
  $below(C, P), right(D, P),$
  where P is a new phantom box.
- $North(\{A_1, \ldots, A_n\}, \{B_1, \ldots, B_m\})$:
  $\forall i \in [1..n], j \in [1..m] : north(A_i, B_j)$
  Note: $North(A, B)$ is equivalent to $North(\{A\}, \{B\})$. The rules are analogous for $South$, $West$ and $East$.
- $Alignment(ExampleGroup, TopToBottom)$:
  $\forall i \in [1..n-1] : north(B_i, B_{i+1})$
  where $ExampleGroup$ is equivalent to $\{B_1, \ldots, B_n\}$

- $Alignment(ExampleGroup, LeftToRight)$:
  $\forall i \in [1..n-1] : west(B_i, B_{i+1})$
  where $ExampleGroup$ is equivalent to $\{B_1, \ldots, B_n\}$
  The rules are analogous for types $BottomToTop$ and $RightToLeft$.

The example layout description presented in section III-E can be transformed to core statements as follows:

- $left(Author, Paper)$
- $left(Paper, Reviewer)$
- $above(Paper, Conference)$
- $west(CoAuthor, Author, Start)$
- $south(CoAuthor, Author, End)$

V. Layout Algorithm

The overview of the layout creation process is depicted in figure 2. The layout definition is first transformed into core language statements according to the rules given in section IV. Another set of core language statements are generated from the model itself: These express widely used diagram arrangement practices.

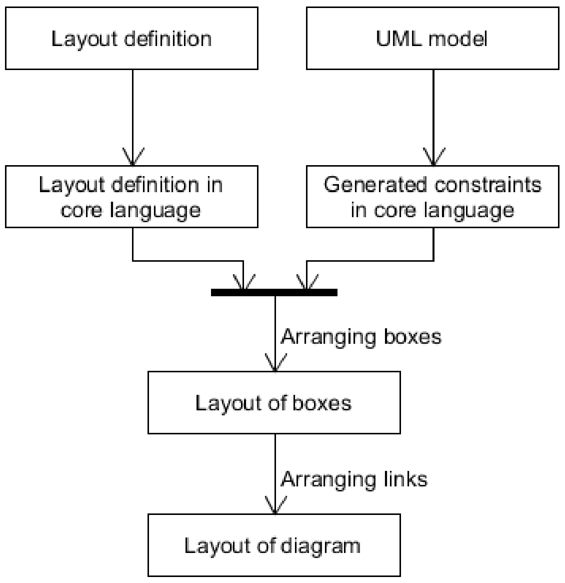

Fig. 2. Overview of the layout algorithm

The core language statements are processed by the box arrangement algorithm first, which calculates the positions of boxes as points on a grid. These results are then converted into a more fine-grained grid, where the boxes also have sizes. This finer grid is used to draw the line of each link.

A. Autogenerated Statements

Beside the constraints given by the user, some statements are generated automatically. This step is particularly important in the case when the user does not define any restrictions on the layout of the diagram. Autogenerated statements can be deleted afterwards if they produce conflicts during the arrangement. There are two types of default statements: those that are generated from special types of links and those that rely only on the boxes and the navigability between boxes using links. Currently the first set only consists of generating statements automatically from generalization type links:

- Link $A \leftarrow B$ is a generalization, where A is the parent and B is the child: $north(A, B)$

The other set of default statements are generated by examining the connectivity graph of the diagram. This graph consists of nodes which represent boxes and a pair of directed edges for each link and statement between two nodes. We detect the connected components of this graph and add constraints that ensure that they will be laid out in disjoint parts of the diagram.

For example, if we have a diagram with boxes: A, B, C, D, E and the following links: $A - B, C - D, C - E$, then there will be two isolated groups: A, B and C, D, E. The generated statements will be the following: $west(A, C)$, $west(A, D)$, $west(A, E)$, $west(B, C)$, $west(B, D)$ and $west(B, E)$.

All of the generated so-called default statements are tagged deletable, so if the arrangement of boxes fails due to conflicts in statements, we just delete a high complexity default statement and try again the arrangement. The complexity of a default statement takes into account the type of the statement (`above` is more complex than `north`, etc.) and the place of generation of the statement (statements generated from links are more complex than those generated from the detection of isolated groups). The generated statements for the previously mentioned example are the following:

- $north(Conference, MainConference)$
- $north(Conference, Workshop)$

B. Layout of Boxes

In this part of the algorithm the boxes are treated as points with no width and height. Therefore only two properties of a box *(A)* are important: its horizontal position on the grid *(A.X)* and its vertical position *(A.Y)*. Our aim is to compute these coordinates.

The earlier defined semantics of the box-related core language statements can be transformed into the following (in)equalities:

- $north(A, B): \quad A.Y - B.Y \geq 1$
- $east(A, B): \quad A.X - B.X \geq 1$
- $south(A, B): \quad -A.Y + B.Y \geq 1$
- $west(A, B): \quad -A.X + B.X \geq 1$
- $above(A, B): \quad A.Y - B.Y = 1$
  $A.X - B.X = 0$
- $right(A, B): \quad A.Y - B.Y = 0$
  $A.X - B.X = 1$
- $below(A, B): \quad A.Y - B.Y = -1$
  $A.X - B.X = 0$
- $left(A, B): \quad A.Y - B.Y = 0$
  $A.X - B.X = -1$

These (in)equalities give a special linear programming problem that consists of simple difference constraints. The system of difference constraints of the statements in the example mentioned earlier are the following:

- $left(Author, Paper)$:
$$\begin{aligned} Author.Y - Paper.Y &\leq 0 \\ -Author.Y + Paper.Y &\leq 0 \\ Author.X - Paper.X &\leq -1 \\ -Author.X + Paper.X &\leq 1 \end{aligned}$$
- $left(Paper, Reviewer)$:
$$\begin{aligned} Paper.Y - Reviewer.Y &\leq 0 \\ -Paper.Y + Reviewer.Y &\leq 0 \\ Paper.X - Reviewer.X &\leq -1 \\ -Paper.X + Reviewer.X &\leq 1 \end{aligned}$$
- $above(Paper, Conference)$:
$$\begin{aligned} Paper.Y - Conference.Y &\leq 1 \\ -Paper.Y + Conference.Y &\leq -1 \\ Paper.X - Conference.X &\leq 0 \\ -Paper.X + Conference.X &\leq 0 \end{aligned}$$
- $north(Conference, MainConference)$:
$$-Conference.Y + MainConference.Y \leq -1$$
- $north(Conference, Workshop)$:
$$-Conference.Y + Workshop.Y \leq -1$$

Instead of applying the simplex method, we can interpret this system of difference constraints from a graph-theoretic point of view [4]. The nodes of the graph are the coordinates of boxes and an extra start node. There are edges from the start node to every other node with a weight of zero. The other weighted edges are generated from the above inequalities:

- $Paper.Y \xrightarrow{0} Author.Y$
- $Author.Y \xrightarrow{0} Paper.Y$
- $Paper.X \xrightarrow{-1} Author.X$
- $Author.X \xrightarrow{1} Paper.X$
- $Reviewer.Y \xrightarrow{0} Paper.Y$
- $Paper.Y \xrightarrow{0} Reviewer.Y$
- $Reviewer.X \xrightarrow{-1} Paper.X$
- $Paper.X \xrightarrow{1} Reviewer.X$
- $Paper.Y \xrightarrow{-1} Conference.Y$
- $Conference.Y \xrightarrow{1} Paper.Y$
- $Paper.X \xrightarrow{0} Conference.X$
- $Conference.X \xrightarrow{0} Paper.X$
- $Conference.Y \xrightarrow{-1} MainConference.Y$
- $Conference.Y \xrightarrow{-1} Workshop.Y$

The Bellman–Ford algorithm can be used to determine the shortest paths from the start node to all others. The costs of these paths give the solution of the system of difference constraints [4].

If the set of core language statements is contradictory, the graph will have a negative circle, which is detected by the Bellman–Ford algorithm. If there are still autogenerated statements in the system, we delete some of them as described in section V-A, and rerun the graph algorithm. Otherwise the user constraints are already contradicting and that triggers an error message.

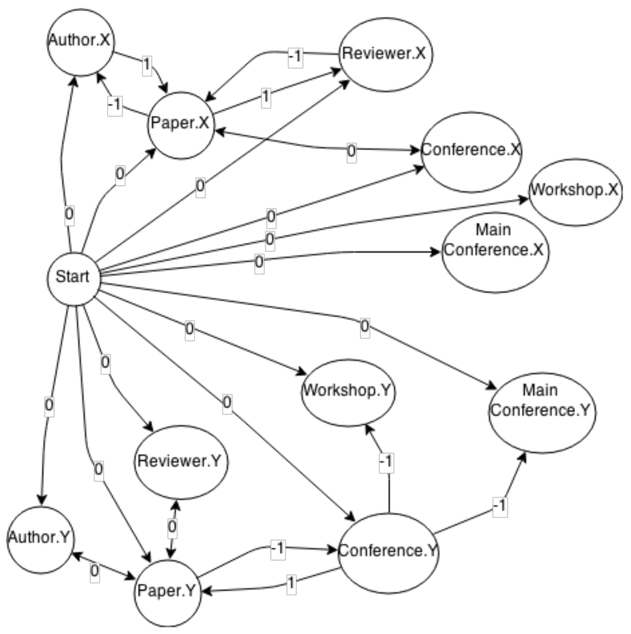

Fig. 3. The complete graph for the the Bellman–Ford algorithm

As mentioned in [4], this method generates a minimal solution in the sense of minimizing $(max\{x_i\} - min\{x_i\})$. In our case it will minimize the maximum distance between boxes, which thus means that there will be no irrelevant space and that the diagram itself will be compact.

Finally there may be boxes that are arranged on each other (Bellman–Ford gave the same coordinates to them). First we fix the current layout of not overlapping boxes by adding $north$ and $east$ statements on every non-overlapping box pairs depending on their coordinates. For example, if box A's coordinate is $(0, 2)$ and box B is at $(0, 4)$ then a $north(B, A)$ statement will be added. Then we try to resolve the problem of overlapping boxes by adding new statements on two boxes with the same coordinate and re-running the Bellman–Ford's algorithm. With this technique we either can produce a diagram with no overlapped boxes or raise an error to the user that we could not design a layout without boxes being overlapped.

C. Layout of Links

The algorithm in the previous section assigns coordinates to the boxes on a grid such that boxes are points and the distance of neighboring boxes is 1. In order to lay out links, boxes need to have dimensions and free space is needed between them. We achieve this by the following steps:

- For each box we count the link ends that will be connected to that box. Let n be the maximum of these numbers.
- All box coordinates are multiplied by $2n + 2$. These will be the coordinates of the top-left corner of the boxes.
- The size of each side of the boxes is $n + 2$.

The result of this is that boxes are squares, all corners are on grid points and each side of a box contains n inner grid points serving as places to connect the link ends. Furthermore, there are exactly n rows (or columns) of free grid points between any two neighboring boxes.

Our main goal is to provide a highly readable UML diagram, therefore we decided to use vertical and horizontal lines because they are easier to read than curves or diagonals [2]. The algorithm lays out links one by one. Drawing a link between the start box and the target one is a typical path finding problem. The links follow the horizontal and vertical lines of the grid. The search space is the directed graph where the vertices correspond to the following grid points:

- Points that are not covered by boxes (ignoring the phantom boxes).
- If there is a $north$, $east$, $south$ or $west$ statement for the start of the link, then the inner grid points on the requested side of the start box.
- If there is no such statements then the inner grid points of all sides of the start box.
- Inner grid points on the side(s) of the target box according to analogous rules.

The start vertex is the center point of the start box and it may not be a grid point. The graph vertices on the side(s) of the target box are the goals.

Each grid line outside the boxes creates two directed edges in this graph. Additional edges lead from the start vertex to the vertices on the perimeter of the start box. If some links have already been laid out, the grid lines matching these links and the grid points where these links turn are removed.

We suppose that the ends of the reflexive links defined by the user statements are on different sides of the box. If a reflexive link does not have such statements, we automatically generate them.

The line of each link is found by an optimal path finding algorithm. The cost of a line p is computed in the following way:

$$\begin{aligned} cost(p) &= c_1 \cdot length(p) + \\ &\quad c_2 \cdot turns(p) + \\ &\quad c_3 \cdot crossings(p) \end{aligned}$$

where c_1, c_2, c_3 are appropriate positive real constants, $turns(p)$ is the number of turning points on the line, $crossings(p)$ is the number of the points where this line crosses a line found earlier.

We would like to use an A^* algorithm to find an optimal path [8]. However, A^* cannot be directly applied on the graph defined above. Namely, this algorithm always works over an edge-weighted directed graph where each weight is greater than a positive threshold. However, the cost of a link shows that the vertices (the grid points) also have weights and the weight of the edges going out from the start vertex is zero. Fortunately, our primary graph can be transformed into a suitable edge-weighted directed graph so that every path going out from the start vertex of this secondary graph corresponds to the lines going out from the start vertex of the previous graph. This secondary graph is the representation graph of the A^* algorithm.

The vertices of the representation graph represent the grid lines that are not in the boxes and that are not occupied by links found earlier. Additionally, the central point of the start box is the start vertex of this graph. The goal vertices are the grid lines connected to the goal box. Generally each grid point generates twelve directed edges that join the grid lines connecting to the very grid point: from north to east, from north to south, etc. If an edge is straight (it leads from east to west or from north to south, etc.), its weight is c_1. If it has a curve, then its weight is c_1+c_2. However, if a link found earlier goes straight through a grid point, only the perpendicular edges remain and their weight is $c_1 + c_3$. If the link turns in the grid point, then this grid point does not generate edges in the representation graph. Moreover, there are extra edges from the start vertex to the vertices of the grid lines connected to the start box and their weight is $2c_1$. Therefore the representation graph is an edge-weighted graph and its weights are greater than $c_1/2$.

If the sequence $< start, v_1, \ldots, v_k >$ is the trace of a link from the start vertex to the goal grid point v_k in the former graph where the grid line n_i leads from v_i to v_{i+1} for each $i \in \{1, \ldots, k-1\}$, then the corresponding path of this link in the representation graph (where the vertices are the grid lines) is the sequence $< start, n_1, \ldots, n_{k-1} >$. This sequence is the reflection of the original trace of a link. We will use the following notation:

$$ref(< start, v_1, \ldots, v_k >) =< start, n_1, \ldots, n_{k-1} > .$$

It is obvious that the sum of the weights of the edges of $ref(p)$ ($cost(ref(p))$) is equal to $cost(p)$ for each line p.

Since the representation graph is finite, the A^* algorithm always terminates: either with the solution path or with failure. The cost of the solution path will be optimal if the A^* algorithm uses an admissible heuristic function, i.e. it is less than or equal to the optimal cost of the remaining path from the current vertex to any goal vertices.

Before defining the heuristic function, some functions, mapping from the vertices to the real numbers, is going to be defined in the primary graph (where the vertices are the grid points). Consider an arbitrary vertex v and the goal vertices. The goal vertices may be around the perimeter of the target box or on its single side. Let us find the goal vertex t that is closest to the vertex v. This goal vertex is unique. The notation $manhattanDistance(v)$ denotes $|v.X - t.X| + |v.Y - t.Y|$. Now consider the minimum length lines between the vertex v and the vertex t regardless the boxes and the links found earlier. Among these lines, select those which have minimal number of turns. Let $turnsRemaining(v)$ be this minimal number, which is either 0 or 1.

It is easy to see that there is no shorter line from the grid point v to the target box than the $manhattanDistance(v)$, and the $turnsRemaining(v)$ is a lower bound on the actual number of turns.

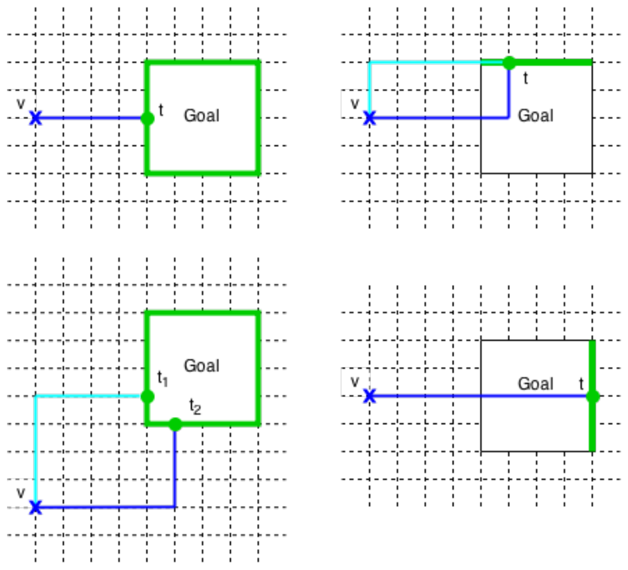

Fig. 4. Computation of the heuristic function of the A^* algorithm. The blue lines give lower bounds on the length and number of turns. The green lines represent the sides including the goal vertices.

The heuristic function h of the A^* algorithm maps from the grid lines (these are the vertices of the representation graph). For all grid lines n, if the ending grid point of this grid line is v, $h(n) = manhattanDistance(v) + turnsRemaining(v)$ and additionally $h(start) = 0$ where $start$ is the start vertex. The $h(start)$ is obviously less than or equal to the cost of the line of each link from the start box to the target one. $h(n)$ also gives a lower bound on the cost of the remaining optimal path from every vertex n derived from a grid line. Namely, this path must be the reflection of the minimal cost line $< v, v_1, \ldots, v_k >$ from the vertex n to any goal vertices in the former graph, thus

$$\begin{aligned} h(n) &= manhattanDistance(v) + turnsRemaining(v) \\ &\leq cost(< v, v_1, \ldots, v_k >) \\ &= cost(ref(< v, v_1, \ldots, v_k >) \end{aligned}$$

and it follows that the heuristic function is admissible.

The graph search that determines the line of a link might fail if the free space between boxes is not enough. If this happens, the coordinates of the boxes are scaled such that the box sizes and the distance of neighboring boxes are multiplied by two. The shortest path algorithm for this link then starts again.

VI. Implementation

A. Embedding in Java

The language is implemented as a Java API because it is an extension to *txtUML*, a domain specific language for executable UML modeling, embedded in Java.

1) Diagram Definition: A diagram is defined as a Java class that extends `Diagram`. It can contain group and phantom definitions and exactly one nested class extending `Layout`, whose annotations define the layout of the diagram:

```
class ExampleDiagram extends Diagram {
  // Group and phantom definitions

  // Layout statements
  class ExampleLayout extends Layout {}
}
```

The boxes and links shown on the diagram are elements of a *txtUML* model (classes and associations in case of a class diagram, for example) and therefore are all represented by different Java classes. These Java classes are used to reference the model elements to be shown on the diagram.

2) Group and Phantom Definitions: As groups and phantoms are not part of a *txtUML* model, they must be defined in the diagram layout description. Groups are nested Java classes extending `NodeGroup`, phantoms are subtypes of `Phantom`. The mandatory `Contains` annotation on groups specifies their members, while the optional `Alignment` annotation might be used to set an alignment of its elements, as the following example shows:

```
class ExampleDiagram extends Diagram {
  @Alignment(AlignmentType.TopToBottom)
  @Contains({Box1.class, Box2.class, Box3.class})
  class ExampleGroup extends NodeGroup {}

  class ExamplePhantom extends Phantom {}

  class ExampleLayout extends Layout {}
}
```

3) Layout Statements: All other statements of the layout language specified in section III are implemented as Java annotations which can be applied to the nested class extending `Layout`. With the exception of the unique `TopMost`, `RightMost`, `BottomMost` and `LeftMost` statements, these annotations are repeatable so they might be used more than once:

```
class ExampleDiagram extends Diagram {
  @Diamond(top = Box1.class, right = Box2.class,
      bottom = Box3.class, left = Box4.class)
  @North(val = Box5.class, from = {Box6.class,
      Box7.class, Box8.class})
  @North(val = Link1.class, from = Box9.class)
  @Priority(val = Link2.class, prior = 10)
  class ExampleLayout extends Layout {}
}
```

4) Example: The following example is the same as the one in section III-E, now using the Java API presented in this section.

```
class ExampleDiagram extends Diagram {
  @Row({Author.class, Paper.class, Reviewer.class})
  @Above(val = Paper.class, from = Conference.class)
  @Show({MainConference.class, Workshop.class})

  @West(val = CoAuthor.class, from = Author.class,
      end = LinkEnd.Start)
  @South(val = CoAuthor.class, from = Author.class,
      end = LinkEnd.End)
  class ExampleLayout extends Layout {}
}
```

B. Papyrus Diagram Generation

The framework of *txtUML* uses Papyrus to represent the visualization of the model with the layout process presented in this paper. We are introducing its functioning through the flowchart in figure 5 for better understanding.

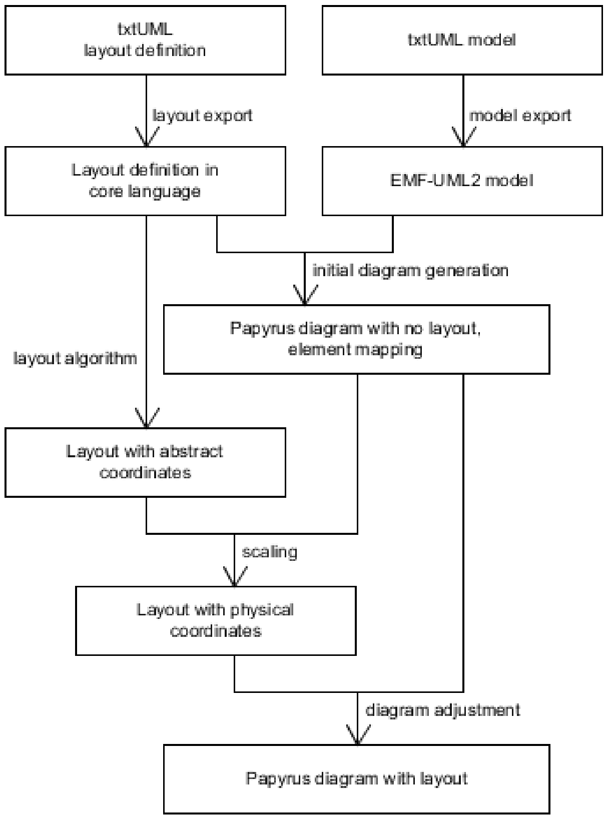
txtUML layout definition
txtUML model
layout export
model export
Layout definition in core language
EMF-UML2 model
initial diagram generation
Papyrus diagram with no layout, element mapping
layout algorithm
Layout with abstract coordinates
scaling
Layout with physical coordinates
diagram adjustment
Papyrus diagram with layout

Fig. 5. Visualization process of txtUML models

1) First, the *txtUML model* is exported to an EMF-UML2 model. This model contains neither layout information nor diagrams. The export functionality is part of the *txtUML* framework.
2) Second, the specific *txtUML layout definition* mentioned above is analyzed and exported to the core language (see section IV).
3) Next, a mapping is created from the elements in the layout language to the EMF-UML2 elements. With these elements in hand, a Papyrus diagram can be established. However, this diagram is not arranged yet.
4) The layout algorithm described in section V arranges the elements based on the layout statements. As a result, we get boxes on the algorithm's uniform integer grid, and the routes of links represented as a list of coordinates on the same grid.
5) On the already established diagram the elements now have their sizes based on their content. With the use of these values the abstract coordinates can be transformed to suite the Papyrus diagrams coordinate system. This transformation scales the boxes based on the maximal box sizes on the diagram.
6) Finally, the elements on the Papyrus diagram are modified according to the new coordinates.

As mentioned in item 5 above, the coordinates provided by the layout algorithm are needed to be scaled up. Another issue is that the algorithm uses a cartesian coordinate system where signed values increase to the right and upwards, unlike computer graphics coordinate systems where unsigned values increase to the bottom and to the right. To sum up, the following transformations are made on the points:

- Flip the ordinate: negate all y coordinates.
- Move the origin to the upper left: This can be easily done by finding the farthest boxes in the negative directions and subtract those values from all coordinates.
- Scale up the grid in order to make room for the boxes: The scale values are defined as maximum box width and height on the Papyrus diagram to make sure that the boxes will not touch each other.

Papyrus uses the Graphical Modeling Framework (GMF) of Eclipse [6], which is a tool for developing graphical editors based on GEF (Graphical Editing Framework) and EMF (Eclipse Modeling Framework). Hence manipulating the diagrams can be done with these frameworks. The frameworks massively support the *command pattern*, so most of the functionality can be accessed through commands and requests. During the implementation, the following GMF and GEF commands and requests were used:

- *ChangeBoundsRequest* changes the bounding box of a box. In GEF graphical elements can be moved or scaled with this request.
- *SetConnectionAnchorsCommand* is used to set the anchors of a link on the starting and ending box.
- *SetConnectionBendpointsCommand* should be applied on a link to specify its route.

With the use of the above mentioned techniques and technologies we have the capability to arrange the elements on our Papyrus diagram, based on the data got from the layout algorithm. As an example for generated Papyrus diagram, see figure 1 in section III.

VII. Evaluation

If the layout algorithm produces a diagram, it is guaranteed to satisfy all constraints given by the user. There are two cases when no diagram is generated:

- The constraints given by the user are contradicting: In this case the layout algorithm stops with an error message naming the boxes being in conflict.
- The user constraints are underspecified so that the algorithm places boxes on top of each other and the heuristics described at the end of section V-B fail to provide a solution: In this case an error message names the overlapping boxes, so that user can add constraints on them.

The following limitations apply for the current version of the algorithm:

- Only flat diagrams are handled, no box-in-box hierarchy is allowed.
- The box sizes on the generated diagram will be equal. The current version of the algorithm is not suitable for laying out diagrams with considerably different box sizes.
- The boxes on the generated diagram will be on a grid. Either two boxes are in the same row (column) or they are in different rows (columns).

In order to evaluate our solution, we have collected 13 class diagrams of different sizes and layout styles from independent sources on the web, and reproduced their layout using the diagram description language. The detailed description of the experiment and its results are published in a technical report [12]. The results presented here were measured using the 0.1.1 version of the txtUML toolset.

We have measured how much the generated diagrams differ from the original ones. The average error, relative to the diagram sizes, was 11.3% in case of a first approximation and 8.4% in case of more accurate layout descriptions. For comparison, the Papyrus autolayout algorithm, which, of course, is not supposed to reproduce the original layout, generated layouts with 50.6% of error for the same diagrams.

The layout descriptions of the first approximation contain 0.54 statements per box on average and 0.93 statements per box in case of the accurate descriptions. For small diagrams (up to 6 boxes) the initial descriptions were created within a minute. The layout description of the largest diagram with 95 boxes took half an hour to define.

The running time of the layout algorithm and the Papyrus diagram generation was between 1-3 seconds for small and medium sized diagrams on a laptop with 1.8 GHz CPU and 4 GB RAM at 1600 MHz. In case of large diagrams, the bottleneck was the layout of links, which took 130 seconds for the largest diagram. This performance problem is expected to be fixed in an upcoming version of the toolset. The memory consumption was rather stable, around 23 MBs as peak value.

VIII. Summary

In this paper we have presented a textual layout language for flat diagrams. The semantics of the layout statements have been defined by translating them to a core language and specifying the core statements using constraints on the coordinates of the diagram elements. We have shown how to apply shortest path algorithms to compute the layout of boxes and links, based on the constraints.

The most important technical aspects of the implementation were also discussed.

Acknowledgment

We dedicate this paper to the memory of Dr. Sándor Sike, our colleague, who recently passed away and was the leader of the Model Driven Development Research Group.

We would like to thank Ericsson for the financial support of this research.

References

[1] Papyrus. http://wiki.eclipse.org/Papyrus.
[2] Scott W. Ambler. *The Elements of UML 2.0 Style*. 2005.
[3] Karl-Friedrich Böhringer and Frances Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 43–51. ACM, 1990.
[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, chapter 24.4 Difference constraints and shortest paths. Third edition, 2009.
[5] Gergely Dévai, Gábor Ferenc Kovács, and Ádám Ancsin. Textual, executable, translatable UML. In *14th International Workshop on OCL and Textual Modeling*, 2014.
[6] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. Papyrus: A UML2 tool for domain-specific language modeling. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 361–368. Springer, 2010.
[7] Ovidiu Gheorghieş. MetaUML: Tutorial, Reference and Test Suite. http://metauml.sourceforge.net/, 2006.
[8] P. Hart, N.J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. System, Man and Cybernet*, 4, (2):100–107, 1968.
[9] Weiqing He and Kim Marriott. Constrained graph layout. In *Graph Drawing*, pages 217–232. Springer, 1997.
[10] Nicolas Kielbasiewicz. The TikZ-UML Package. http://perso.ensta-paristech.fr/~kielbasi/tikzuml/, 2012.
[11] Hugo AD Nascimento and Peter Eades. User hints for directed graph drawing. In *Graph Drawing*, pages 205–219. Springer, 2002.
[12] Dávid János Németh, András Nagy, Balázs Gregorics, András Dobreff, Gábor Ferenc Kovács, Gergely Dévai, and Tibor Gregorics. Reproducing UML class diagrams using txtUML. Technical report, Eötvös Loránd University, Faculty of Informatics, 2015.
[13] Oracle. The Java Tutorial, Laying Out Components Within a Container. https://docs.oracle.com/javase/tutorial/uiswing/layout/.
[14] Helen C Purchase, Jo-Anne Allder, and David A Carrington. Graph layout aesthetics in UML diagrams: user preferences. *J. Graph Algorithms Appl.*, 6(3):255–279, 2002.
[15] K Sugiyama, S Tagawa, M Toda, MJ Carpano, MR Garey, DS Johnson, P Eades, S Whitesides, P Eades, NC Wormald, et al. Methods for Visual Understanding of Hierarchical Systems. *IEEE Trans. Syst. Man Cybern.*, 25:261–295, 1994.
[16] Till Tantau. The Ti*k*Z and PGF Packages. Institut für Theoretische Informatik, Universität zu Lübeck, 2012.
[17] Markus Voelter. Embedded software development with projectional language workbenches. In *Model Driven Engineering Languages and Systems*, pages 32–46. Springer, 2010.